Universidade de Aveiro
Departamento de
Eletrónica, Telecomunicações e Informática
2015

Luís Filipe
Abade Rodrigues

Códigos de Correção de Erros para Sistemas de Comunicação por Luz Visível

Error Correcting Codes for Visible Light Communication Systems

**Luís Filipe
Abade Rodrigues**

Códigos de Correção de Erros para Sistemas de Comunicação por Luz Visível

Error Correcting Codes for Visible Light Communication Systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Luís Filipe Mesquita Nero Moreira Alves, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Carlos Miguel Nogueira Gaspar Ribeiro, Professor Adjunto do Departamento de Engenharia Electrotécnica da Escola Superior de Tecnologia e Gestão de Leiria.

o júri

Presidente	**Professor Doutor Paulo Miguel Nepomuceno Pereira Monteiro** Professor Associado do Departemento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro
Arguente	**Professora Doutora Mónica Jorge Carvalho Figueiredo** Professora Adjunta do Departamento de Engenharia Electrotécnica da Escola Superior de Tecnologia e Gestão de Leiria.
Orientador	**Professor Doutor Luis Filipe Mesquita Nero Moreira Alves** Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos

Durante a realização desta dissertação foram muitos os contributos recebidos das mais variadas formas. Assim, gostaria de deixar os meus agradecimentos aos que mais diretamente contribuiram neste trabalho.

Em primeiro lugar gostaria de agradecer ao meu orientador, Professor Doutor Luis Filipe Mesquita Nero Moreira Alves por me ter proporcionado trabalhar num tema inovador e por todo o acompanhamento aconselhamento ao longo deste trabalho

Não menos importante, deixo um especial agradecimento ao meu coorientador, Professor Doutor Carlos Miguel Nogueira Gaspar Ribeiro por todas as críticas construtivas e por toda a ajuda prestada que se revelou fundamental na realização desta dissertação.

Ao Instituto de Telecomunicações de Aveiro pelo espaço e materiais disponibilizados, sem os quais seria impossível a realização deste trabalho.

Ao meu amigo Luís Duarte pela força e determinação transmitidas, por todos os conhecimentos partilhados e por todos os momentos que passámos ao longo dos últimos anos. Ao meu amigo Pedro Rodrigues por todos os bons conselhos transmitidos e por toda a ajuda que demonstrou ao longo deste tempo.

Aos meus colegas de laboratório, Filipe, André, Miguel, Nuno, Pedro e Gonçalo por toda a ajuda prestada e pelos bons momentos de descontração.

Aos meus pais, Paulo Rodrigues e Helena Rodrigues, pela confiança, pelos conselhos, pela compreensão e por todo o apoio que sempre prestaram. À minha irmã, Ana Rodrigues por toda a paciência demonstrada.

Aos meus avós, Abel, Maria do Rosário, Rui e Licína por todo o apoio que sempre demonstraram.

Por último, mas não menos importante, à Marisa por estar sempre do meu lado, pela força que me transmitiu nos momentos mais complicados, pela compreensão, pela paciência e por acreditar em mim.

A todos, muito obrigado.

Resumo

Ao longo dos últimos anos o número de utilizadores de redes sem fios tem aumentado. Até ao momento, a tecnologia RF (Radio Frequência) dominado este segmento. No entanto, a saturação nessa região do espectro eletromagnético exige tecnologias alternativas para redes sem fios

Recentemente, com o crecimento do mercado da iluminação LED (Díodo Emissor de Luz), as Comunicações por Luz Visível têm atraído as atenções dos investigadores. Em primeiro lugar, é uma fonte de luz eficiente para aplicações de iluminação. Em segundo lugar, o LED é um dispositivo que é facilmente modulado e com grande largura de banda. Por último, permite combinar iluminação e comunicação no mesmo dispositivo, ou seja, permite a implementação de sistemas de comunicação sem fios altamente eficientes.

Um dos aspetos mais importantes num sistema de comunicação é a sua fiabilidade quando sujeitos a canais com ruído. Nestes cenários, a informação recebida pode vir afetada de erros. Para garantir o correto funcionamento do sistema, é muito comum o uso de um codificador de canal. A sua função é codificar a informação a ser enviada para melhorar a performance do sistema. O uso de Códigos de Correção de Erros é muito frequente permitindo anexar informação redundante aos dados originais. No recetor, a informação redundante é usada para recuperar possíveis erros na transmissão.

Esta dissertação apresenta os passos da implementaç ão de um Codificador de Canal para VLC. Foram consideradas várias técnicas tais como os códigos Reed-Solomon e os códigos Convolucionais, Interleaving (Bloco e Convolucional), CRC e Puncturing. Foi efetuada uma análise das características de cada técnica a fim de avaliar quais as mais apropriadas para o cenário em questão. Numa primeira fase, vários modelos foram implementados em Simulink a fim de simular o comportamento dos mesmos em diferentes cenários. Mais tarde os modelos foram implementados e simulados em blocos de hardware. Para obter resultados de uma forma mais rápida, foram elaborados modelos de co-simulação em hardware. No final, diferentes técnicas foram combinadas para criar um Codificador de Canal capaz de detetar e corrigir errors aleatórios e em rajada, graças ao uso de códigos Reed-Solomon em conjunto com técnicas de Interleaving. Adicionalmente, usando o algoritmo CRC, após o processo de descodificação, o sistema proposto é capaz de identificar possíveis erros que não puderam ser corrigidos.

Abstract

Over the past few years, the number of wireless networks users has been increasing. Until now, Radio-Frequency (RF) used to be the dominant technology. However, the electromagnetic spectrum in these region is being saturated, demanding for alternative wireless technologies.

Recently, with the growing market of LED lighting, the Visible Light Communications has been drawing attentions from the research community. First, it is an efficient device for illumination. Second, because of its easy modulation and high bandwidth. Finally, it can combine illumination and communication in the same device, in other words, it allows to implement highly efficient wireless communication systems.

One of the most important aspects in a communication system is its reliability when working in noisy channels. In these scenarios, the received data can be affected by errors. In order to proper system working, it is usually employed a Channel Encoder in the system. Its function is to code the data to be transmitted in order to increase system performance. It commonly uses ECC, which appends redundant information to the original data. At the receiver side, the redundant information is used to recover the erroneous data.

This dissertation presents the implementation steps of a Channel Encoder for VLC. It was consider several techniques such as Reed-Solomon and Convolutional codes, Block and Convolutional Interleaving, CRC and Puncturing. A detailed analysis of each technique characteristics was made in order to choose the most appropriate ones. Simulink models were created in order to simulate how different codes behave in different scenarios. Later, the models were implemented in a FPGA and simulations were performed. Hardware co-simulations were also implemented to faster simulation results. At the end, different techniques were combined to create a complete Channel Encoder capable of detect and correct random and burst errors, due to the usage of a RS(255,213) code with a Block Interleaver. Furthermore, after the decoding process, the proposed system can identify uncorrectable errors in the decoded data due to the CRC-32 algorithm.

Contents

List of Figures v

List of Tables ix

Acronyms xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Context . . . 2
- 1.3 Methodology . . . 2
- 1.4 Dissertation Structure . . . 3
- 1.5 Contributions . . . 3

2 Visible Light Communications 5
- 2.1 Optical Wireless Communications History . . . 6
- 2.2 LED History . . . 7
- 2.3 Motivation . . . 9
- 2.4 Actual Context . . . 10
- 2.5 Optical Sources . . . 11
- 2.6 Optical Detectors . . . 13
- 2.7 Optical Filters . . . 14
- 2.8 Modulation Schemes . . . 14
  - 2.8.1 OOK . . . 14
  - 2.8.2 OFDM . . . 15
  - 2.8.3 DCO-OFDM . . . 15
  - 2.8.4 U-OFDM . . . 15
  - 2.8.5 ACO-OFDM . . . 16
- 2.9 Forward Error Correction . . . 16
  - 2.9.1 Reed-Solomon Codes . . . 16
  - 2.9.2 Convolutional codes . . . 17
  - 2.9.3 Turbo Codes . . . 17
  - 2.9.4 Concatenated codes . . . 18
  - 2.9.5 Interleaving . . . 18
  - 2.9.6 Puncturing . . . 19
- 2.10 System architecture . . . 19
  - 2.10.1 Overview . . . 19
  - 2.10.2 Data Link Layer . . . 19
  - 2.10.3 Emitter Physical Layer . . . 21

2.10.4 VLC Channel . . . 22
2.10.5 Receiver Physical Layer . . . 23
2.11 Tools and Devices for System Implementation . . . 24
2.11.1 Spartan-6 FPGA SP605 Evaluation Kit . . . 24
2.11.2 Xilinx System Generator . . . 26
2.11.3 Asynchronous Architecture . . . 26
2.12 Concluding Remarks . . . 28

3 Channel Coding **29**
3.1 VLC Channel Noise Sources . . . 29
3.2 Error patterns . . . 32
3.3 Bit Error Rate analysis . . . 33
3.4 Reed-Solomon Codes . . . 34
3.4.1 BER Analytical Estimation for Reed-Solomon Codes . . . 35
3.5 Convolutional Codes . . . 36
3.5.1 BER Analytical Estimation for Convolutional Codes . . . 39
3.6 CRC . . . 40
3.7 Interleaving . . . 41
3.8 Puncturing . . . 43
3.9 Concluding Remarks . . . 43

4 System Design Considerations **45**
4.1 Simulink Models . . . 45
4.2 Synchronous Implementation . . . 57
4.2.1 Reed-Solomon codes . . . 59
4.2.2 Convolutional codes . . . 63
4.2.3 Interleaving . . . 66
4.2.4 CRC-32 . . . 69
4.2.5 Hardware co-simulation . . . 72
4.3 System Constraints . . . 79
4.4 Concluding Remarks . . . 85

5 Implementation and Final Results **87**
5.1 Channel Encoder model . . . 87
5.2 FPGA Utilization . . . 97
5.3 System Performance . . . 98

6 Conclusions **101**
6.1 Conclusions . . . 101
6.2 Future Work . . . 102

Bibliography **103**

A CRC-32 model **107**

B System Constraints MatLab script file **117**

C MatLab scripts **121**

D Asynchronous Implementation FSMs **125**

List of Figures

2.1 Visible light in electromagnetic spectrum . . . 5
2.2 Photophone . . . 6
2.3 White LED with a phosphor layer . . . 8
2.4 Haitz' Law . . . 9
2.5 Planar LED Structure . . . 13
2.6 LED output power saturation at high current values . . . 13
2.7 RS(N,K) codeword structure . . . 16
2.8 Convolutional encoder with shift-registers . . . 17
2.9 Turbo code encoder . . . 18
2.10 RSCC block diagram . . . 18
2.11 VLC system architecture overview . . . 19
2.12 MAC Layer description . . . 20
2.13 VLC ACO-OFDM Emitter Physical Layer . . . 21
2.14 VLC DCO-OFDM Emitter Physical Layer . . . 21
2.15 VLC ACO-OFDM Receiver Physical Layer . . . 23
2.16 VLC DCO-OFDM Receiver Physical Layer . . . 24
2.17 FPGA architecture . . . 24
2.18 Spartan-6 FPGA SP605 Evaluation Kit . . . 25
2.19 Asynchronous architecture - Emitter . . . 26
2.20 Asynchronous architecture emitter main block . . . 27
2.21 Asynchronous architecture - Receiver . . . 27
2.22 Asynchronous architecture receiver main block . . . 28

3.1 LOS propagation model . . . 30
3.2 Normalized power/unit wavelength for optical wireless spectrum and ambient light sources . . . 31
3.3 Gilbert-Elliott Model with two states . . . 32
3.4 RS(N,K) codeword structure . . . 34
3.5 Reed-Solomon encoder hardware implementation . . . 34
3.6 Reed-Solomon decoder diagram block . . . 35
3.7 M-ary input, M-ary output, symmetric memoryless channel . . . 36
3.8 Convolutional Encoder . . . 37
3.9 Convolutional Encoder K=3, n=2, k=1 . . . 37
3.10 State machine of a convolutional code K=3, n=2, k=1 . . . 38
3.11 Trellis diagram of a convolutional code K=3, n=2, k=1 . . . 38
3.12 CRC-N codeword . . . 40
3.13 Interleaving example . . . 41
3.14 Block interleaver matrix . . . 42

3.15 Convolutional interleaver and deinterleaver 42
3.16 Convolutional interleaving output 43

4.1 BSC simulink model 45
4.2 BSC Simulink model simluation results 46
4.3 Gilbert-Elliott simulink model, for burst error simulation 46
4.4 Gilbert-Elliott Simulink model simluation results 47
4.5 Reed-Solomon simulink model in presence of random errors 48
4.6 RS(255,K) with random errors Simulink model simluation results 49
4.7 RS(255,213) with random errors Simulink model simluation results 50
4.8 Reed-Solomon simulink model in presence of burst errors 50
4.9 RS(255,213) with burst errors Simulink model simluation results 51
4.10 Reed-Solomon with Block Interleaving simulink model in presence of burst errors 52
4.11 RS(255,213) + Block Interleaving with burst errors Simulink model simluation results 53
4.12 Convolutional codes simulink model for random errors 53
4.13 Convolution Encoder for K=7, R=1/2 and Generators 171_8 and 133_8 54
4.14 Convolutional codes with random errors Simulink model simluation results . 54
4.15 Convolutional codes simulink model in presence of burst errors 55
4.16 Convolutional codes with random errors Simulink model simluation results . 56
4.17 Convolutional codes with an Convolutional Interleaver simulink model in presence of burst errors 56
4.18 Convolutional codes + Convolutional Interleaving with burst errors Simulink model simluation results 57
4.19 System Generator random error channel model 57
4.20 System Generator Token block 58
4.21 BSC System Generator simulation results 59
4.22 System Generator Reed-Solomon model 60
4.23 RS(255,213) System Generator model simluation results 63
4.24 System Generator Convolutional synchronous model 64
4.25 Convolutional code R=1/2, K=7 System Generator model simluation results 65
4.26 System Generator Convolutional Interleaving synchronous model 66
4.27 System Generator Block Interleaving synchronous model 68
4.28 System Generator CRC-32 model 69
4.29 System Generator CRC-32 emitter synchronous model 70
4.30 Hardware implementation of a serial CRC generator 70
4.31 System Generator CRC-32 receiver synchronous model 72
4.32 BSC co-simulation System Generator model 73
4.33 Generic model for hardware co-simulation data acquisition 74
4.34 Hardware co-simulation results for the a BSC 75
4.35 RS(255,213) co-simulation System Generator model 76
4.36 ChipScope Pro waveforms of input and input data bits, showing an misalignment between signals 76
4.37 Convolutional code R=1/2, K=7 co-simulation System Generator model . . . 77
4.38 Performance of the RS(255,213) co-simulation System Generator model . . . 78
4.39 Performance of the Convolutional code R=1/2, K=7 co-simulation System Generator model 78

4.40 Frame structure 79
4.41 VLCLighting encoder and decoder schemes 80
4.42 Performance of QPSK modulation over an AWGN channel 80
4.43 Performance of RS(255,213) and Convolutional (R=1/2, K=7) 81
4.44 Reed-Solomon(255,K) codes efficiency vs BER performance 82
4.45 FER vs system efficiency 83
4.46 FER vs system efficiency for a frame size=K-CRC 84

5.1 Asynchronous architecture of the proposed Channel Coding scheme 87
5.2 Asynchronous architecture - Data Generator 88
5.3 Asynchronous architecture - Channel Encoder 88
5.4 Asynchronous architecture - CRC-32 encoder 89
5.5 Asynchronous architecture - RS encoder 91
5.6 Asynchronous architecture - Block Interleaver 91
5.7 Asynchronous architecture - Error Generator 92
5.8 Asynchronous architecture - Sync Output 92
5.9 Asynchronous architecture - Channel Decoder 93
5.10 Asynchronous architecture - Block Deinterleaver 93
5.11 Asynchronous architecture - Reed-Solomon Decoder 94
5.12 Asynchronous architecture - CRC-32 decoder 95
5.13 Asynchronous architecture - CRC result analysis 96
5.14 Asynchronous architecture - High data rate Sync Output block 98
5.15 Clock Counter Block 99
5.16 Frame Error Rate system performance 100

List of Tables

2.1 Comparison between LED an LD characteristics 12

3.1 International Visibility range and Attenuation coefficient in the Visible Waveband for Various Weather conditions 31
3.2 Rate 1/2 maximum free distance code 39
3.3 Coding gain for soft-decision Viterbi decoder 40

5.1 Device Utilization Summary - Channel Encoder 97
5.2 Device Utilization Summary - Channel Decoder 98
5.3 System data rate results 99

Acronyms

ACO-OFDM Asymmetrically Clipped Optical OFDM

ADC Analog-to-Digital Converter

APD Avalanche Photodiode

ARQ Automatic Repeat Request

AWGN Average White Gaussian Noise

BER Bit Error Rate

BLER Block Error Rate

BSC Binary Symmetric Channel

CDMA Code-Division Multiple Access

CP Cyclic Prefix

CRC Cyclic Redundancy Check

DAB Digital Audio Broadcast

DCO-OFDM DC-Optical-OFDM

DLL Data Link Layer

DMT Discrete Multitone

DVB Digital Video Broadcast

ECC Error-Correcting Codes

EMI Electromagnetic Interference

FDMA Frequency-Division Multiple Access

FEC Forward Error Correction

FER Frame Error Rate

FFT Fast Fourier Transform

FOV Field of View

FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FSO	Free Space Optics
GF	Galois Field
IR	Infra-Red
IFFT	Inverse Fast Fourier Transform
ISI	Intersymbol Interference
IrDA	Infrared Data Association
LAN	Local Area Network
LASER	Light Amplification by Stimulated Emission of Radiation
LD	LASER Diodes
LED	Light Emitting Diode
LFSR	Linear Feedback Shift Register
LOS	Line-of-sight
MAC	Medium Access Control
NRZ	Non-Return-to-Zero
ODAC	Optical Digital-to-Analog Converter
OFDM	Orthogonal Frequency Division Multiplexing
OOK	On-Off Keying
OWC	Optical Wireless Communications
PAPR	Peak-to-Average Power Ratio
PLC	Power-Line Communications
PSD	Power Spectral Density
PWM	Pulse-Width Modulation
RF	Radio Frequency
RS	Reed-Solomon
RZ	return-to-zero
SDMA	Space-Division Multiple Access
SPAD	Single-Photon Avalanche Diode

SNR	Signal-to-Noise Ratio
SSD	Solid-State Devices
TDMA	Time-Division Multiple Access
U-OFDM	Unipolar-OFDM
UV	Ulta-Violet
VLC	Visible Light Communications
xDSL	Digital subscriber line
WDMA	Wavelength-Division Multiple Access

Chapter 1

Introduction

Over the last years, mobile wireless communications have been taking a prime role in modern society. The growing demand of high rate services such as high-speed Internet access and high-definition television are rapidly saturating the available bandwidth of actual Radio Frequency (RF) systems. Besides the crowded spectrum issue, RF systems have their usage regulated, usually associated to expensive licenses, and presents low data rate capabilities and high energy consumption.

Optical Wireless Communications (OWC) use optical radiation, either Visible Light or Infra-Red (IR), to transmit data through the free-space channel and can benefit from the unregulated spectrum, large available bandwidth, Electromagnetic Interference (EMI) immunity and improved security. With such characteristics, OWC can be an alternative to existing RF systems where free spectrum can be a problem or the use of them is prohibited.

Currently, the illumination market is gradually adopting LED technology. The development of High Brightness white LED makes these devices ideal for lighting systems, assuring energy efficiency, low environmental impact and high luminous output.

Visible Light Communications (VLC) are an emerging field of OWC that exploits the lighting systems to communicate, particularly HB white LED based ones, since these Solid-State Devices (SSD) can be easy modulated at high frequency. VLC systems are susceptible to noise and interference from the free-space channel and design imperfections that degrade system performance, inducing errors in transmission. There are several techniques to mitigate such undesirable effects.

In this dissertation, several Forward Error Correction (FEC) techniques to increase VLC Orthogonal Frequency Division Multiplexing (OFDM) based systems performance. The analysis includes Reed-Solomon codes, Convolutional codes, Interleaving and Puncturing, that were studied, simulated and implemented in an asynchronous architecture. The tools used to develop this work was a Xilinx Spartan®-6 FPGA SP605 Evaluation Kit, Xilinx System Generator®, MatLab® and Simulink®.

1.1 Motivation

Real communication systems are susceptible to noise and interferences, either from the systems' imperfections or from the communication channel, which can corrupt the received data. Designing a communication system in a proper way, identifying the noise sources and overcoming some of the imperfections, can mitigate such undesirable effects. Nonetheless, in

real implementations there will remain some noise in the system. In addiction, the communication channel is not ideal either. Several sources of noise are added to the transmitted signal, which can be modelled depending on its nature. However, these mathematical models can not predict the exact behaviour of the noise, giving only a statistical model instead.

The system performance can be estimated by knowing its parameters such as the modulation scheme and the channel conditions. One of the most relevant parameters is the Signal-to-Noise Ratio (SNR). In 1948, Shannon establish the channel capacity for a given channel of a certain bandwidth and a known SNR. In order to reach the channel capacity, several methods must be used together, such as information encoding, modulation schemes and channel equalisation. Forward Error Correction are a group of techniques to improve system performance, approaching the capacity to Shannon limit.

Currently, VLC systems are in development, achieving recently 3.4 Gb/s using an RGB LED [1] with a technique called Wavelength Division Multiplexing (WDM). Offering such data rates, this technology can be used in many scenarios where RF is the standard and also in situations where RF is prohibited, not recommended or it does not work properly, such as in airplanes and mines.

VLC applications includes indoor and outdoor scenarios, each one having its characteristics. On the one hand indoor scenarios suffers from non-LOS as well as multipath interferences. On the other hand, ambient light and channel attenuation can reduce system performance in outdoor scenarios. The main goal of this dissertation is to evaluate which FEC techniques can effectively reduce bit error rate in an outdoor OFDM VLC system.

1.2 Context

The work developed in this dissertation is a part of a larger project called VLCLighting, which is a collaborative research project being developed in Instituto de Telecomunicações de Aveiro. VLCLighting aims to develop public lighting systems with integrated communication capabilities using VLC, delivering a demonstrator transmitting video and data by the end of 2016. The project also proposes to develop a modular system to offer a real-time test bed to evaluate the performance of different modules, algorithms and optical front-ends.

The proposed system is composed by four distinct modules: Data Link Layer, Channel Encoder, Modulation and optical front-ends. Data Link Layer is responsible for framing and fragmenting the input data, assuring continuous transmission; Channel Encoder, which is studied in detail in this document, implements FEC techniques to provide data protection to noise and interference; The modulation block employs QPSK and DCO-OFDM; The emitter optical front-end is an Optical Digital-to-Analog Converter (ODAC) and the receiver uses a photodiode and a transimpedance amplifier.

1.3 Methodology

The work developed in this dissertation is divided in 4 parts: theoretical analysis, Simulink models, System Generator models, hardware co-simulation and Asynchronous Implementation.

The theoretical analysis aims to understand several FEC techniques characteristics which allows to decide what are the best techniques for the present scenario.

Using Matlab Simulink tool, a series of models will be created in order to verify how the FEC schemes work and be able to simulate them in different scenarios. The results will be compared to the expected values in order to validate the models.

The next phase is to transport the simulink models into the System Generator environment. The main goal is to replace Simulink blocks (encoder and decoder blocks) by Xilinx System Generator blocks.

In order to obtain simulation results faster, some of the System Generator models will be converted into hardware co-simulation models.

The final step is to merge several FEC techniques in an asynchronous architecture and evaluate its performance.

1.4 Dissertation Structure

This dissertation is divided in 6 Chapters. The present chapter describes the motivation and context of this work. The following chapters are enumerated bellow:

- Chapter 2 describes the actual context and achievements of the VLC. It also presents an introduction to FEC schemes. At the end of this chapter is given an overview of the used tools for this work.
- Chapter 3 goes into detail on the Channel Coding topic. Why it is important to consider a channel encoder, what are the errors that can occur in the channel, how several FEC schemes works and what are their main characteristics are some of the questions answered in this chapter.
- In Chapter 4 are shown the implemented model in Simulink, as well as in System Generator. This chapter ends with an analysis which studies the best FEC scheme to be implemented in the final implementation.
- The Chapter 5 presents the details of the Asynchronous Channel Encoder implementation as well as its performance.
- Chapter 6 shows the final discussions of this work. At the end, some future work guidelines are presented.

1.5 Contributions

This work contributed with a paper titled "VLCLighting - A Collaborative Research Project on Visible Light Communication", published in the 10th Conference on Telecommunications ConfTele 2015, realized in Aveiro, Portugal [2].

Chapter 2

Visible Light Communications

The need of communication appeared in the early civilizations. The first telecommunications systems were mainly VLC and included fire, smoke or sunlight to send messages over great distances. In the 19th century, telecommunications systems based on electrical signals such as telegraph and telephone, required a physical wire connection between terminals. Wireless radio systems were developed in the beginnings of 20th century by Guglielmo Marconi. RF systems such as Wifi, Bluetooth, mobile phones, among others, are widely spread in modern society. The high demand for wireless services rapidly consumes the available RF spectrum, requiring the development of alternative solutions. In the last few decades researchers have been focused in development of OWC technology which can present a viable solution to some of RF drawbacks.

Visible Light Communications are a field of OWC that uses the visible part of electromagnetic spectrum, between 400 and 800 THz, as seen in Figure 2.1, to transmit data. This systems makes use of the existing lighting fixtures to provide both illumination and communication which improves power efficiency. Recently, in 2011, the IEEE VLC standard (802.15.7) was approved, provided to the engineers a solid ground to development commercial VLC systems. Despite the relevance of this work, the transmission rates are significantly lower than the VLC capabilities. By using different modulation schemes such as OFDM instead of OOK could provide higher transmission rates and lower error probabilities.

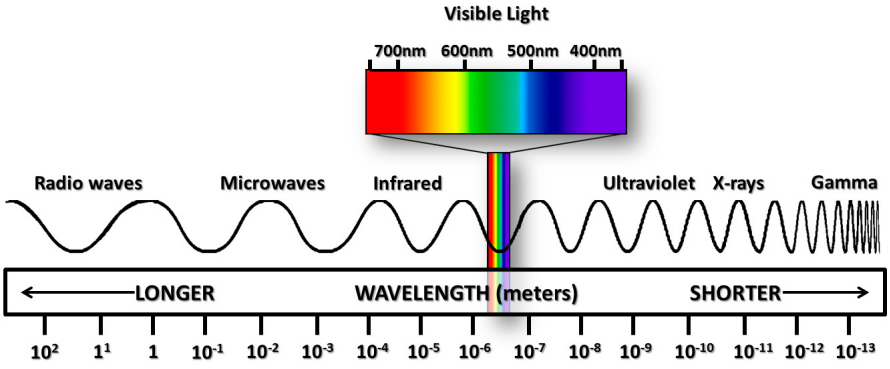

Figure 2.1: Visible light in electromagnetic spectrum [3]

This chapter begins to introduce a brief history of OWC and LED, followed by the most important achievements in VLC, giving to the reader a perspective of its context. Then, an overview of commonly used optical sources and detectors are presented, as well as the modulation schemes for VLC, giving emphasis to OFDM and its variants. In the sequence of the dissertation main topic, several FEC techniques are mentioned, along with their main characteristics. The VLCLighting system architecture is presented in the last section of this chapter, explaining the role of each component.

2.1 Optical Wireless Communications History

One of the first Free Space Optics (FSO) communication systems dates back to 800 BC when ancient Greeks and Romans used fire beacons for signalling. In 405 BC, during the day, sunlight was reflected by mirrors with an instrument denominated Heliograph. Later, in 280 BC, the Lighthouse of Alexandria was built and around 150 BC the American Indians were using smoke for communication purposes. Another application of visible light for signalling was using semaphores in sea navigation, in 1790 [4] [5].

The first VLC technological achievement was presented by Alexander Graham Bell, called Photophone, in Figure 2.2, achieving distances up to 200m. His system makes use of sunlight reflected by a mirror which vibrates according to a sound pressure wave. The reflected light was then received by a selenium photocell, inside a parabolic mirror, connected to a sound transducer [5] [6]. Due to the crudity of materials used and the inconstancy of sunlight, this device was never adopted [4].

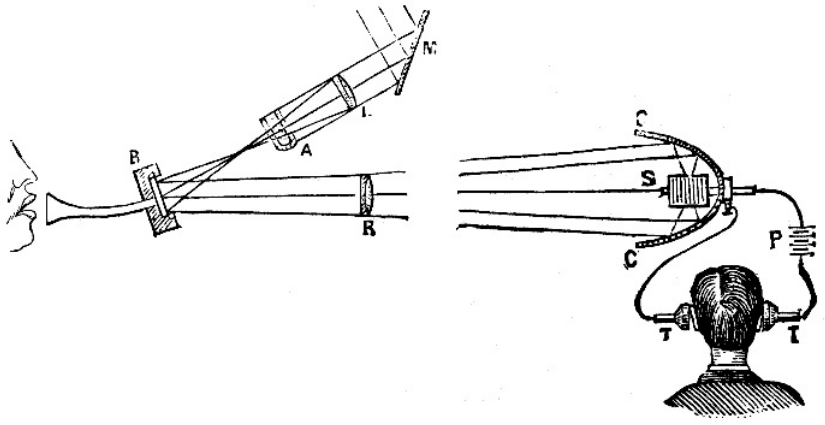

Figure 2.2: Photophone [7]

The OWC would not suffers great advances until the 1960s, when Light Amplification by Stimulated Emission of Radiation (LASER) radiation and LASER diode were invented, followed by the development of low-loss optical fibres, in the 1970s, the optical fibre amplifier in the 1980s and the invention of in-fibre Bragg grating in the 1990's. These inventions are on the basis of actual Internet Infrastructure [6]. Also in the 1960s Light Emitting Diode (LED) was invented. These two new light sources, LASER and LED, led to new experiences such as TV signal transmission over 48 Km using GaAs LED by MIT Lincolns Laboratory, in 1962; voice transmission through a He-Ne LASER over 190 Km, in May 1963, and the first TV-

over-laser demonstration in March 1963. In 1970, Nippon Electric Company (NEC) built the first full-duplex LASER link to commercial traffic, in Japan, over 14 Km [4]. The first indoor OWC system was proposed in 1979, by Gfeller and Bapst, and uses diffuse near-IR radiation to interconnect a cluster of terminals located in a room to a common cluster controller [6].

In 1997 Joseph Kahn and John Barry presented their work in Wireless Infrared Communications [8] which aims to exploit IR radiation for high-speed short-range wireless communications. It describes a detailed study of the IR channel characteristics, including path losses and multipath effects as well as several noise sources. Besides, they presented emitter and receiver designs that maximizes the SNR. The analysis included several modulation schemes performance such as OOK, pulse-position modulation (PPM) and subcarrier modulation. Furthermore, it presents techniques to include multiple users in the transmission. At the end, an experimental system using OOK is described using a translucent plastic diffuser in front of a LD, achieving a transmission rate of 50 Mb/s. The measured error probability of this experimental system at a distance of 4.4m is $10^{-}7$.

Although the repeated successful experiences in OWC, these systems, at the time, were not useful for many reasons: first, existing communication systems were satisfactory for that time; second, the development required to much resources for a reliable system; third, the atmosphere conditions limits the usage of OWC systems, for example in the presence of fog; fourth, some systems require a complex tracking system, which does not exists at the time [4].

Recently, with the proliferation of solid-state lightning devices such as LEDs, particularly white LEDs, VLC comes as a solution to some disadvantages of RF communications [6].

2.2 LED History

With the evolution of wired and RF communications, and the lack of a stable light source easy to modulate, VLC research seams a waste of time. However, in 1920, Oleg Vladimirovich Losev, a young and extremely talented scientist working as a technician in several Soviet radio laboratories, observed light emission on silicon carbide (SIC) crystal rectifier diodes used in radio receivers, when a current passed through them. In 1907, H. J. Round had already observed the same effect, but the two discoveries have no relation. Years later, in 1927, he writes a paper entitled "Luminous carborundum [silicon carbide] detector and detection with crystals" in which he established the current threshold for the onset of light emission from the point of contact between a metal wire and a SIC crystal and recorded the emitted spectrum. In his 16 papers, he provided a comprehensive study of LEDs and outlined their applications. In his work, he studied the cold (non-thermal) nature of emission, the current-voltage characteristic, the thermal dependence and the frequency response. He also explained how the emission process works, based in Einstein quantum theory. Losev was the only one at his time to understand the potential of LEDs for telecommunications, writing, on 31 December 1929 [9]:

> "The proposed invention uses the known phenomenon of luminescence of a carborundum detector and consists of the use of such a detector in an optical relay for the purpose of fast telegraphic and telephone communication, transmission of images and other applications when a light luminescence contact point is used as the light source connected directly to a circuit of modulated current."

Although his research and vision could change the OWC, he died young with his work unfinished. Later, in the 1950s, Infrared and Green LEDs were patented as well as SIC visible light LEDs [10]. It was only in 1962 that the first practical LED was produced, by Nick Holonyak Jr., emitting red light, made of GaAsP crystals. The performance of these first LEDs was very poor, about 0.1 lumens/Watt, compared to a typical incandescent bulb, with a performance around 15 lumens/Watt. For that reason, LEDs were primarily used as indicators, which requires a low amount of light.

The next step occurred in 1968 with the introduction of nitrogen in the GaP crystal (used in green LEDs) and in the GaAsP (red LEDs), in 1971. With this innovation GaAsP:N LEDs performance increases to 1 lumens/Watt and two new colours were now available: orange and yellow. In the early 1980s, red LEDs, due to usage of AlGaAs, achieved 2 to 10 lumens/Watt, which enabled LEDs to compete with incandescent lights, for instance, in automotive taillights and outdoor moving-message panels. In 1990, C. P. Kuo and his co-workers developed a new way to produce yellow LED, made of AlInGaP, which increased the performance, comparable to the best AlGaAs red LEDs. By varying the alloy composition, it was possible to produce high brightness radiation from red through green.

Although the advances in LED development, this devices were not yet suitable for lightning because of the lack of white colour. At the beginning of the 1990s, although it was possible to make efficient red and green LEDs, the blue colour was missing, making it impossible to produce white light. These devices have been available since the 1970s, made of SIC crystals, however, their performance was very low, about 0.1 lumens/Watt [11]. The first blue high bright LED was presented in the beginnings of 1990s, by Shuji Nakamura, Isamu Akasaki and Hiroshi Amano [12]. They used GaN which, at that time, was very difficult to produce in large quantities, and worse, create a p-type layer in this material was virtually impossible. The three researchers was convinced this is the best material for blue LEDs and choose it instead of zinc selenide, the one everyone claimed to be the most promising.

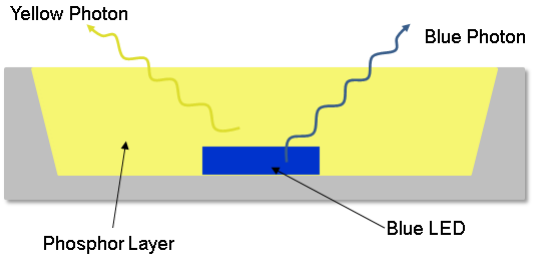

Figure 2.3: White LED with a phosphor layer

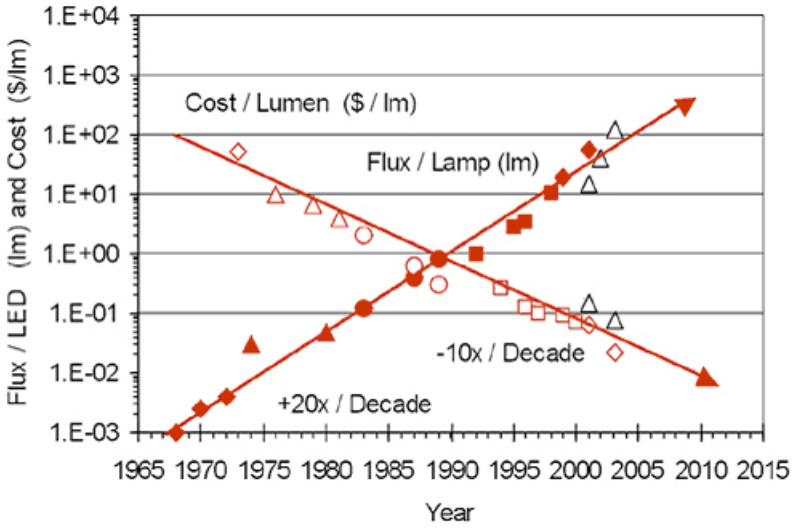

Figure 2.4: Haitz' Law [13]

After the discovery of blue LEDs, it was just needed a small step to finally produce white light. One way to do this is just add phosphor to a blue LED: once phosphor is excited by blue wavelengths, it emits yellow radiation (located in red and green spectrum), as seen in Figure 2.3. The other way is to group 3 LEDs: red, green and blue. Since white LEDs were manufactured, the applications did not stop growing: LCD-screens backlit, flashlights, camera flash and home and public lighting, among others [12].

An overview of evolution of LED is described by Haitz' Law, shown in Figure 2.4, which states that LED technology increases its luminous flux by 20 times per decade and decreases its cost per lumen by a factor of 10 per decade. This law was proposed in 1999, and presented in 2000, and has been updated over the years, being included, in 2010, white LEDs. Although this is a predictive law, it has been used to study possible new markets and it is often compared to Moore's Law [14].

2.3 Motivation

Mobile communications are taking an important role in modern society, from private to business and safety applications. Broadband wireless services such as HDTV, computer network applications, mobile phones, video conferencing, high-speed Internet access, suffered a fast growth over the past decades, leaving radio frequency spectrum fully occupied. This has demanded alternatives to RF communications. On the other hand, although Radio Frequency communications provide mobility in indoor and outdoor scenarios, over small and large coverage areas, there are some disadvantages, such as bandwidth regulation, security issues, high installation costs and low data rates [4].

Visible Light Communications comes as a solution to some of the RF disadvantages [4] [5] [15]:

- License-free bandwidth, since it is visible light, the same we use to lighting our world;

- Almost unlimited bandwidth, every room can use all the available spectrum which is much larger than RF spectrum;
- High data rates due to LED high modulation capability and high available bandwidth;
- Absence of electromagnetic interference;
- Hazard free, compared to RF communications, since light is part of our nature;
- Improved security, because light cannot penetrate walls;
- Lower power consumption, either for using the same light for both communication and illuminating purposes, or the possibility to concentrate power transmission in Line of Sight communications.

Despite these advantages, VLC has a few drawbacks which can be obstacles to implement it in real scenarios [4] [5] [15]:

- Higher susceptibility to noise due to atmospheric conditions, like clouds, fog, rain and snow;
- Interferences from background light, such as sunlight radiation and traditional luminaries;
- Narrow coverage;
- VLC is in an initial development state.

2.4 Actual Context

In the past fifteen years, many efforts have been made to develop VLC systems with progressively higher data rates and reliability hand-in-hand with increasing the number of applications for this technology.

One of the first well known projects using VLC was the OMEGA (Home Gigabit Access) project, which started in 2006. The main goal of OMEGA was set a global standard for ultra broadband home area networks, achieving 1Gb/s combining different existing communication systems, such as, Power-Line Communications (PLC), RF and VLC. The project idea was to simplify network access, making them as easy to use as an electricity socket, ending with coverage problems and also reducing the required wiring. Thus, users can easily set-up an home network to access to high-bandwidth information and communication services. The project successful demonstrated a VLC system broadcasting four HD videos simultaneously [16].

At the same time, Intelligent transportation systems (ITS) using VLC were proposed. Several applications in this field can be exploited: improved safety, by integrating communication capabilities in LED traffic lights or between vehicles to warning the driver about dangerous situations, or in some cases, the vehicle could take actions to prevent an accident; location and route informations, provided either by the road infrastructure, to locate the vehicle in a city, or by the vehicle, to send vehicle actual location to the network (useful for instance, in bus traffic control and scheduling); electronic tolling, discarding the currently

used RF modules. In 2007, VIDAS (VIsible light communications for advanced Driver Assistance Systems) project started in Instituto de Telecomunicações and developed a functional traffic light with VLC demonstrator.

One of the most important steps was taken in 2011 with the approval of VLC standard, the IEEE 802.15.7.

Fraunhofer Heinrich Hertz Institute also developed a functional plug and play VLC product based on HB white LED. Their system is bidirectional, adapting dynamically the rate of the transmission. It uses Discrete Multitone (DMT) modulation to achieve 500Mb/s in ideal scenarios. It guarantees data rates up to 120Mb/s over a distance of 100m and up to 100Mb/s in non-LOS over 3m [17].

In 2008, D-Light, later renamed as pureLiFi, started to develop a VLC for high-speed data communication using commercial LED infrastructures, presenting their first product in 2011, achieving 5Mb/s for each link in a full duplex scheme, over a range of up to three meters. The Li-1st uses white LEDs for the downlink, providing simultaneously illumination and communication, and IR for the uplink. Recently, in March 2015, Li-Flame was presented, increasing the data rate to 10Mb/s for downlink and 10Mb/s for uplink, offering full mobility and multiple user support.[18]

In a successful attempt to provide a positioning system for indoor scenarios, since current systems such as GPS (Global Positioning System) can not work in such conditions, Philips Lighting presented in May 2015 an indoor positioning system using VLC and LED luminaries. Consuming only 50% the previous lighting system, the costumers of an hypermarket can be guided directly to discounts with less than one meter of precision, using their smartphone camera as an optical receiver [19].

an hypermarket can now guide directly the costumers to discounts,

Recently, VLC was proposed to complement RF technology in 5G-PPP, due to some VLC desirable characteristics for 5G, such as increased spectral efficiency, improved coverage and energy efficiency. While RF serves outdoor scenario, VLC could be a viable solution to meet the 5G requirements[20]. In a recent White Papper from NetWorld2020 group [21], three scenarios were identified where VLC could be a complementary technology:

- Cellular OWC where optical access points offers localized high data rate transmissions in heterogeneous networks. In this scenario, localization services were also considered;
- Optical hotspots in indoor scenarios where VLC-over-fibre could be implemented;
- Smartphone short-range communication could provide highly directional communication giving alternatives to the present technologies, achieving higher data rates.

2.5 Optical Sources

To achieve high-speed rates in VLC first we need a light source that can be modulated fast. For instance, traditional incandescent lamps are not suitable to high-speed communications because its light flux depends on filament temperature, which varies slowly. The most commonly used light sources are LEDs and LASER Diodes (LD). While LEDs are used mostly in short range applications in indoor scenarios, mainly in hybrid Line-of-sight (LOS) or non-LOS, LD, due to their highly directivity beam profile, ideal for LOS, are used in long distance communication. Both LEDs and LDs rely on the electronic excitation of semiconductor materials for their operation. These devices do not emit thermal radiation, unlike the

incandescent bulbs. Other properties of both, includes small size devices, low forward voltage and drive current and excellent brightness in visible wavelengths and the option of emission of a single or a range of wavelengths [4]. Although LEDs and LDs have some characteristics in common, LDs requires a more complex driver with temperature stabilization, variations in temperature leads to changes in spectral emission and the maximum emitted power allowed is limited due to eye safety. However, LDs can be used in higher modulation rates than LEDs, making them suitable for high-speed data communication [15].

The table 2.1 summarizes the differences between LEDs and LDs:

Characteristic	LED	LD
Optical spectral width (nm)	25 - 100	10^{-5} - 5
Directionality	Broad	Narrow
Modulation Bandwidth	∼kHz to ∼MHz	∼MHz to ∼GHz
Special circuitry required	None	Threshold and Temperature required
Eye safety	Considered eye safe	May need to be rendered eye safe
Reliability	High	Moderate
E/O conversion efficiency	10-20%	30-70%
Cost	Low	Moderate to high

Table 2.1: Comparison between LED an LD characteristics

The generation of light in these solid state devices is due to the transition of an electron in high energy state to a lower energy state. The energy difference of the two levels is released in a radiative or non-radiative process. In the first one, the energy is released in form of light, while in the second the energy is converted to heat.

The interaction between an electron and a photon can occur in 3 different ways: photon transmit its energy to an electron in the filled valence band, which is excited to the conduction band; an electron in the filled conduction band can spontaneously return to the valence band and release a photon, in a process called spontaneous radiative recombination, which occurs in LEDs; a photon can be used to stimulate the radiative recombination process, releasing a second photon with same phase as the first one, that is, the two photons are coherent (this is the principle of LASER) [4].

The frequency of the emitted or absorbed photon, f,is related with the difference of the two energy bands, E_1 and E_2, given by equation 2.1 [4]:

$$E = E_2 - E_1 = hf = \frac{hc}{\lambda} \tag{2.1}$$

where h is the Plank's constant, c is the speed of light in vacuum and λ is the wavelength of the photon.

The LED is a semiconductor p-n junction device which, in presence of electronic excitation in the form of a forward bias voltage, gives off spontaneous optical radiation, in a process called spontaneous radiative recombination. The energy given by electronic excitation transfer electrons in the valence band (stable state) to the conduction band (unstable state) that spontaneously return to the stable state, emitting energy in form of photons. The emitted radiation can be Ulta-Violet (UV), visible or IR, depending on the band-gap of semiconductor materials [4]. Generally, the usage of longer wavelengths, typically 1330nm and 1550nm, leads to a cheaper optical detectors and GaAs laser diodes. A planar LED structure is represented in Figure 2.5 [15].

In LEDs the process is fairly efficient, so the heat produced in the device is much lower than in incandescent bulbs. Non-radiative recombination occurs when instead of a photon, the recombined electron gives off a phonon (heat) [4]. Since LEDs are p-n junctions, the voltage-current curve is identical to a regular p-n diode, only beginning to conduct from a certain voltage. The relationship between radiated optical power and the current through the device is linear. It becomes non-linear for high current values, introducing distortion, as shown in Figure 2.6.

One important LED characteristic to consider in VLC is its bandwidth, which depends on the injected current, the junction capacitance and the parasitic capacitance. While frequency response increases with forward current, capacitances values are near invariable [4].

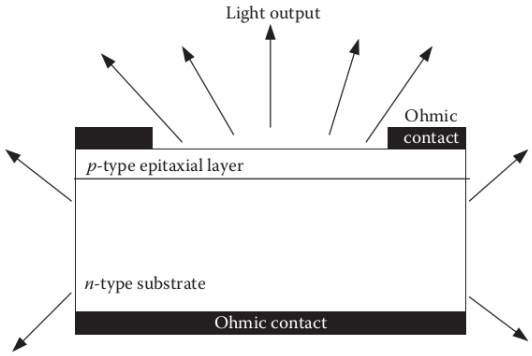

Figure 2.5: Planar LED Structure [4]

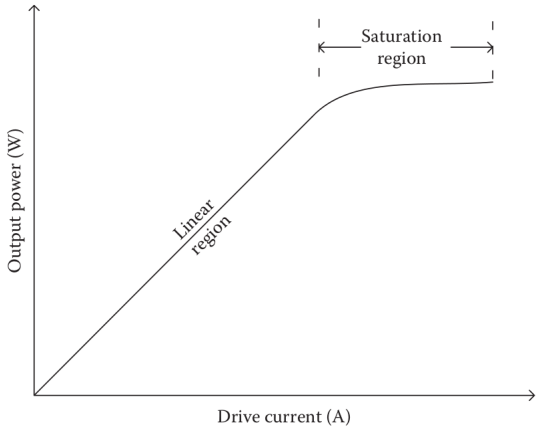

Figure 2.6: LED output power saturation at high current values [4]

2.6 Optical Detectors

The conversion of light into electrical signals is an important aspect in VLC. First of all, it is required a photodetector with high sensitivity and fidelity, high optical to electri-

cal conversion efficiency, large detection area, low noise, short response times, low cost and high reliability. Naturally, many of these characteristic cannot coexist, being necessary some compromise in the choice of a photodetector. There are three main types of devices that convert light into an electrical signal: pin photodiodes, Avalanche Photodiode (APD)s and Single-Photon Avalanche Diode (SPAD).

Due the widespread commercial availability, pin photodiodes and APDs are usually the choice for OWC. Despite pin photodiodes are the simplest ones, either in their structure and employment, those devices have less sensitivity compared to the APDs, which can be 10-15dB higher. Although systems that uses APDs are more robust to losses, they need high bias voltages, exhibit highly non-linear current gain and sensitivity to reverse bias voltage, which could be more vulnerable for ambient light interference. Optical detectors, as well as optical sources, have constraints in frequency, determined by the device capacitance and the resistance presented by the receiver. In indoor OWC, due the fact of high amount of ambient light, large area detectors are more effective to attenuate the interference. This however, increases the capacitance, thus requiring a trade-off between detection area and capacitance and also the implementation of methods to decrease capacitance [15].

2.7 Optical Filters

Optical filters are commonly used in the OWC receivers to minimize the interferences from ambient background radiation, such as sunlight or indoor illumination. The main goal is to create a band-pass filter, allowing, only the incident radiation that transports information, to reach the photodiode [15]. In Visible Light Communication systems based in phosphorescent white LEDs, which as an increased response time due to radiation emitted by the phosphor layer, an optical blue filter is usually included to convert only the blue component of the LED emission. With such technique, the bandwidth can be enhanced to 20-25MHz. However, in this case, only about 50% of optical power is converted to an electrical signal which can decrease the performance of the system if none additional processing is used, such as better modulation techniques or the introduction of Error-Correcting Codes (ECC) [22].

2.8 Modulation Schemes

2.8.1 OOK

One of the most commonly used modulation in VLC is On-Off Keying (OOK) mainly due to its simplicity. Each bit is represented by an optical pulse in a slot time corresponding to a bit duration, wherein the LED is turned on if the transmitted bit is one and switched off in the opposite case, for Non-Return-to-Zero (NRZ) scheme. The alternative is to use return-to-zero (RZ) scheme, reducing in half the bit time. This modulation is widely used in commercial applications such as Infrared Data Association (IrDA) and Fast IR Links operating below 4Mbits/s [4]. It is evident that the brightness of the light emitted decreases about 50% comparing to a light source without communication capabilities, if the data source contains the same number of 0's and 1's. Since LEDs dimming is realized by Pulse-Width Modulation (PWM), OOK is not a suitable modulation to combine lighting and communication features [23]. Other types of modulation such as VPPM (Variable Pulse Position Modulation) combines pulse width (dimming) with its position (at the beginning or at the

end of the time slot depending of the transmitted bit).

2.8.2 OFDM

The VLC Channel multipath characteristic decreases the high-speed systems performance due to the interference caused by the multiple reflections in walls and furniture, etc, or by sudden blockage. In RF communications this problem is also an issue, OFDM is a solution to this problem [24]. This modulation technique is adopted in communication systems such as Digital Audio Broadcast (DAB), Digital Video Broadcast (DVB), Wireless Local Area Network (LAN) and Digital subscriber line (xDSL) [25]. OFDM is a multi-carrier scheme modulation that splits up a high data rate signal into a set of low rate sub-streams in parallel. Thus, the total bandwidth is divided into several sub-carriers, each one carrying a low rate data stream. In the receiver, having multiple sub-carriers simplifies the channel equalization process [26]. Another feature implemented by OFDM is a time guard interval to prevent multipath-induced Intersymbol Interference (ISI), which is a time gap between OFDM symbols, allowing the majority of multipath reflections reaching the receiver without interfering with a new symbol. This time guard is Cyclic Prefix (CP) which preserves the orthogonality between sub-channels [26].

The main blocks of an OFDM modulator are a bit to symbol mapping operation, inverse Fast Fourier Transform (FFT) operation and the CP insertion [27].

For applications where multiple users can access the medium, OFDM brings an issue related with high values of Peak-to-Average Power Ratio (PAPR), due to the superposition of emitted signals from different users. High PAPR values requires a large linear range of amplification in the receiver to ensure communication without distortion, increasing the power required and compromising the system efficiency [26].

Due to imaginary values at IFFT output, this type of modulation is not directly suitable for VLC, since LEDs needs real and positive values to directly modulate the light intensity. To overcome the problem with imaginary values, Hermitian symmetry is imposed on the complex symbols that are feed into IFFT block, thus guaranteeing real values at output [4]. Although this technique is useful for VLC, it doubles the subcarriers required, the first half with data and the second half with the conjugate values of the first half, in reverse order, reducing the bandwidth efficiency to about 50% [28].

2.8.3 DCO-OFDM

To solve the issue caused by OFDM bipolar signal in VLC, a DC-bias is introduced in real time-domain OFDM symbols, shifting the negative values to the positive scale. This technique is known as DC-Optical-OFDM (DCO-OFDM). The disadvantage that comes with this modulation is a higher power consumption due to the DC component, degrading the system efficiency [29]. However, it can be easily employed in scenarios where illumination is required since it presents a constant mean light output value. Furthermore, the DC value is variable making dimming a possibility in this type of modulation.

2.8.4 U-OFDM

In VLC systems which requires higher levels of efficiency than the DCO-OFDM provides, the OFDM symbols must be converted to an unipolar signal, avoiding the use of the DC shift. One way to achieve this is to transmit two OFDM frames wherein the first one contains only

the positive values of the original OFDM signal, leaving the negative values with zero value, and in the second frame the remaining values, the negative ones, are sent with their absolute value, with positive values as zero. This modulation is called Unipolar-OFDM (U-OFDM). Since its needed two frames to transmit one OFDM frame, U-OFDM spectral efficiency is about a half of the spectral efficiency of DCO-OFDM.

A simple way to demodulate the U-OFDM is to subtract the second frame (negative samples) from the first one (positive samples) and then continuing the demodulation as an usually OFDM symbol. The subtracting process doubles the Average White Gaussian Noise (AWGN) variance, reducing the system performance by 3dB.

An alternative to demodulate U-OFDM requires the samples of the two frames to be paired (one from the positive and one from the negative). The receiver must be able to detect, in each pair, which one is the sample that carries valid information, discarding the other one. Comparing a pair of samples, the one with higher amplitude is marked as active. Discarding the inactive sample cancels AWGN associated with it, increasing the system performance [29].

2.8.5 ACO-OFDM

Parallel to U-OFDM, Asymmetrically Clipped Optical OFDM (ACO-OFDM) is a technique to avoid the addition of the DC component, suggested by DCO-OFDM. In ACO-OFDM the data is carried only in the odd frequency OFDM subcarriers, while the even subcarriers have zero value. After the IFFT operation, the real signal is clipped to zero, leaving only the real positive values. Although the clipping process generates noise, this affects only the even frequencies, which does not contains any information. ACO-OFDM, as well as U-OFDM, has a spectral efficiency about a half of the spectral efficiency of DCO-OFDM [29].

2.9 Forward Error Correction

2.9.1 Reed-Solomon Codes

Reed-Solomon (RS) codes are a type of FEC technique and it is probably the most widely used codes in real communication and storage systems. They work by dividing the data stream into blocks of K data elements and then adding w redundant elements to each block. Therefore the code word consists in $N = K + w$ elements, forming a codeword specified as RS(N,K) [30]. These codes are non-binary codes, that is, they are described in terms of symbols, which are elements of finite fields or Galois fields GF. A GF is a finite set of elements that has defined arithmetic operations which produce also elements of the field [25].

The basic idea of RS codes is to generate, at the emitter side, a codeword which contains, in the first field, the K data symbols to be transmitted and in the second field the w parity symbols (Figure 2.7), obtained by computing the remainder of the division of the information polynomial by a generator polynomial [31].

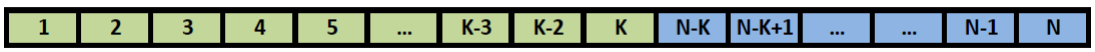

Figure 2.7: RS(N,K) codeword structure

The correction capabilities of RS codes depends on its Hamming distance, related to the

number of redundant elements. The Hamming distance of RS codes is specified by $m+1$ and they can detect up to w symbols in error and correct up to $t = w/2$ symbols.

Due to its strong error correction capability and its simple implementation, RS codes are used in storage systems (hard disks, CD, DVD and barcodes), wireless communications, DVB, xDSL, deep space and satellite communications [30].

2.9.2 Convolutional codes

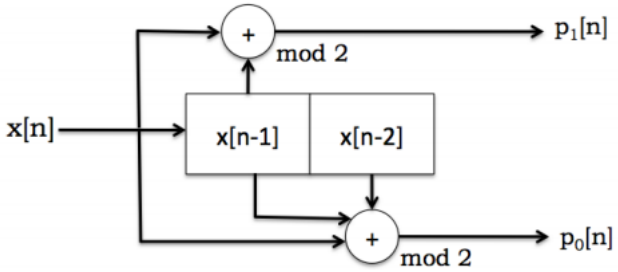

Figure 2.8: Convolutional encoder with shift-registers

As well as RS codes, Convolutional codes are a FEC technique, also widespread in telecommunication systems, such as 802.11 and satellite communications. These codes differ from block codes because the sender only transmits the parity bits, calculated from input data bit. The rate of these codes is $1/r$, where r is the number of parity bits. These codes are suitable for channels with a random error pattern.

The encoding method uses a sliding window, with a predefined length, able to preform a modulo 2 addition of several subsets of bits. This process generates a parity bit from each operation. The window shifts one input bit at the time (see Figure 2.8). The size of the window is called the code's constraint length. As the constraint length increases, one data bit will influence more parity bits: if one parity bit are received with errors, the data can be recovered by other parity bits also affected by the same information. However, longer constraint lengths increases the decoding time.

In the receiver the parity bits must be decoded to recover the original information. One method to do this is called maximum likelihood decoding, determining the nearest valid codeword to the parity bits received. However, for N-bit transmission, there are 2^N possible codeword, making this algorithm inefficient for large sequences [32].

A more efficient way to decoding convolutional codes is using the Trellis structure. Trellis diagram is based on states, evolving over time for each received parity bit. The path through Trellis diagram should decode the received data. Viterbi algorithm makes use of Trellis structures to decode the data in the presence of errors. The key insight in Viterbi decoders is their ability to calculate the most likelihood path, based on calculated Hamming distances for each branch [33].

2.9.3 Turbo Codes

Convolutional Turbo codes consisting of two or more recursive systematic convolutional encoders separated by an interleaver. The resultant codeword is composed by the parity bits,

followed by the data bit (see Figure 2.9). These codes, developed in 1993, were the first class of ECC that can perform close to the Shannon limit [34]. An alternative to Convolutional Turbo codes are Block Turbo Codes. Due to their Bit Error Rate (BER) performance with no additional power requirement, Turbo codes applications includes satellite communications, 3G wireless cellphones, DVB and WMAN [35].

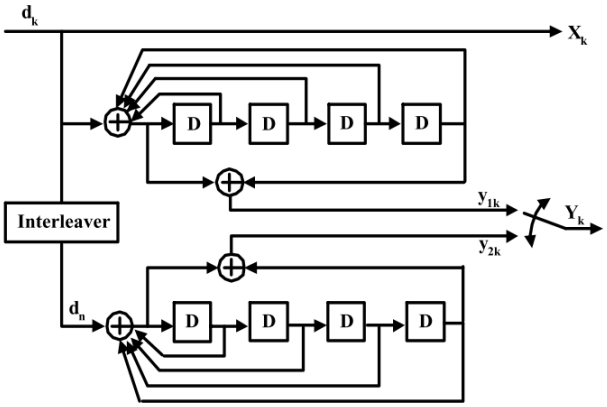

Figure 2.9: Turbo code encoder [36]

2.9.4 Concatenated codes

In communication systems it is commonly used a technique that combines a inner code and a outer code. The usage of a non-binary code for the outer code an a binary for inner code results in a close approach to the channel capacity limit. The popular example, shown in Figure 2.10, is the Reed-Solomon Convolutional Concatenated (RSCC), in which the RS and the convolutional code are outer and inner codes, respectively. In this scheme, the RS is ideal to correct burst errors and the convolutional code corrects spread errors, giving to the system the capacity to handle two different types of errors, improving its reliability [37].

Figure 2.10: RSCC block diagram

2.9.5 Interleaving

The communication channel can generate two main types of errors: random errors and burst error. Convolutional codes are particularly efficient to random error channels. However, this type of channel may sometimes exhibit a burst error characteristic and lead to errors that can not be corrected in the receiver. One way to reduce the impact of burst error is to increase the code capability at expense of high redundancy.

The interleaving technique comes as a solution to this problem, mixing up all the code symbols from different codewords, spreading the burst errors into multiple codewords. Therefore, a simple random error correcting code is sufficient to correct one symbol per codeword. It is evident that interleaving process transforms a burst error channel into a random error one [38].

2.9.6 Puncturing

To increase the bitrate, puncturing techniques are used with convolutional codes to delete some bits in the code bit sequence. In the decoder, the errors introduced by puncturing method are corrected. The advantage of using this method is that it is easy to vary the redundancy of the code depending of the channel quality, using the same decoder [39].

2.10 System architecture

2.10.1 Overview

A typical VLC system is shown in Figure 2.11. A digital data stream is inserted in the system input and it is processed by the Medium Access Control (MAC), where the data is grouped in frames. The MAC is then sent to the Physical Layer to be modulated and transmitted to the Tx front-end: the LED Matrix. The optical receiver front-end is composed by a photodiode and a transimpedance amplifier. The Physical Layer in the receiver demodulates the signal provided by the receiver front-end and the data is sent to the MAC layer to decode the MAC messages, recovering the original digital signal.

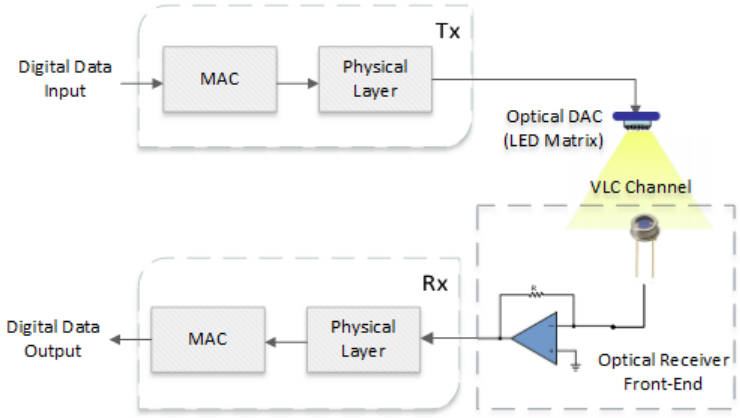

Figure 2.11: VLC system architecture overview

2.10.2 Data Link Layer

The MAC layer is an interface between the Physical layer and the higher layers of the system, adapting the data to the physical system. This layer also handles the translation of physical addresses into addresses used by the higher layer. The main blocks of the MAC Layer are shown in Figure 2.12.

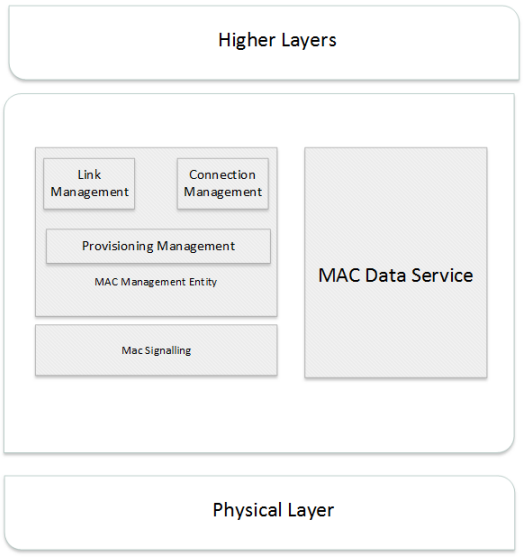

Figure 2.12: MAC Layer description

The block MAC Data Service provides services which can accommodate data in order to work with frames. A frame is a structure with the original data and other parameters, important or optional, such as frame delimiters, source and destination addresses, type of modulation in current frame, sequence number, length of the information, etc. This block is also responsible for adding stuff bits in order to fill all the length of payload field, if the input data is not long enough. In this block, a simple error checksum is inserted to verify, in the receiver, if the frame contains valid information.

There are scenarios where it is useful to have a frame to send parameters of the system, such as modulation type, encryption, bandwidth request and channel quality indicator, to the receiver. This frames are generated in MAC Signalling block [40] which has also the function of generate idle frames to maintain visibility when there is no data to transmit [41]. The signalling frames have no error checksum field.

The Link Management block controls where the receivers are connected physically. For instance, if two LED Matrixes (and the respective Physical Layer) are connected to the MAC block, the last one must know in which LED Matrix is connected the destination device. On the other hand, the Connection Management block translates the address from the upper layer to a MAC address.

If the channel is shared by multiple emitters, there must be a way to prevent two or more transmitters from sending information at the same time, originating collisions and signal degradation. The responsible function for avoiding collisions is the Provisioning Management, which controls the access of the physical layer to the medium.

The collision avoidance can be implemented in the optical domain, by controlling the wavelength of each emitter, also known as Wavelength-Division Multiple Access (WDMA), and Space-Division Multiple Access (SDMA), in which each emitter operates in a restricted space. In the electrical domain there are 3 main techniques to prevent collisions: Time-Division Multiple Access (TDMA), Frequency-Division Multiple Access (FDMA) and Code-Division Multiple Access (CDMA). TDMA reserves a slot time while FDMA reserves frequency bands, for each emitter. Due to the simple implementation, TDMA is a viable solution for unidi-

rectional systems which require a high bit rate, using all the available bandwidth during the time slot.

2.10.3 Emitter Physical Layer

The Physical Layer is responsible for the modulation of the data from the MAC Layer and to implement FEC codes. In Figure 2.13 it is shown the Physical Layer implemented with ACO-OFDM modulation.

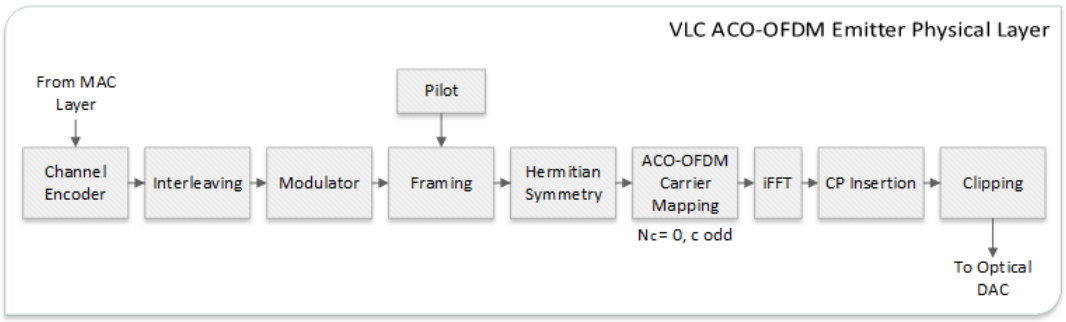

Figure 2.13: VLC ACO-OFDM Emitter Physical Layer

The first step after the MAC Layer process is to implement FEC techniques. This is made in the Channel Encoder implementing an ECC, followed by an interleaver, which is advantageous if the ECC used is a concatenated code. After the FEC calculations, the data bits are translated to symbols in the Modulator, which can be either QAM or QPSK.

The next process is to modulate the data symbols into OFDM. The Framing block makes the distribution of the data subcarriers, pilot subcarriers (for synchronization and channel estimation) and the null subcarriers. After that, Hermitian Symmetry is imposed to generate a real value at the IFFT output. In ACO-OFDM, the even subcarriers are set to zero, as explained before. After the IFFT operation, a CP is inserted and the signal is clipped to zero.

An alternative scheme is using DCO-OFDM modulation. The differences to the previous scheme is that all the subcarriers are used and not only the odd ones, and at the end, instead of clipping the signal, a DC-Bias value is added to preserve the negative values at the IFFT output, as observed in Figure 2.14.

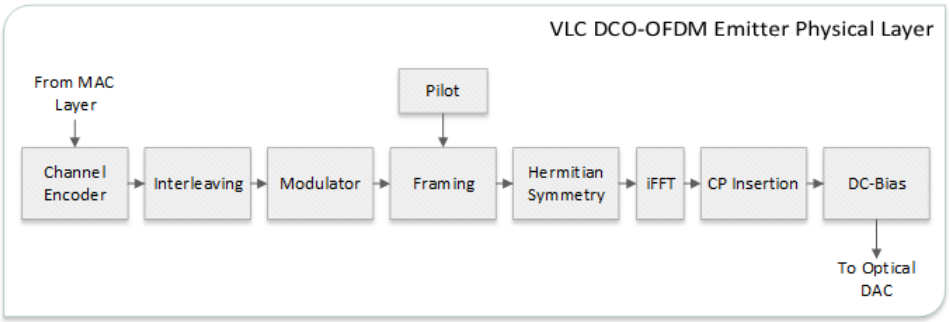

Figure 2.14: VLC DCO-OFDM Emitter Physical Layer

The output to the ODAC is a binary output with N bits: the bit p switches on 2^p LEDs, where $p = \{0, 1, ..., N-1\}$. It is clear that the LED Matrix is composed by $2^N 1$ LEDs, giving 2^N levels of light intensity. Sometime it is used a technique to make the matrix symmetrical which consists in duplicating the LEDs for each bit, thus doubling the number of LEDs required. Thus, the output light intensity between each level doubles, reducing the SNR.

2.10.4 VLC Channel

Due to the noise induced by the VLC channel it is worth to take some time in its study. The power losses in the channel are due to optical path loss and multipath dispersion. Depending if the system is LOS or non-LOS, these two channel components will have different impact in the transmission. In LOS systems, the reflections represents a minor element in the transmission power losses, on the other hand non-LOS, in indoor communications for instance, the received optical power is highly affected by a multipath dispersion component. The main tools to study the behaviour of the received optical power are several propagation models to predict the path loss.

Considering null reflections, in the LOS indoor scenario, only the path loss is taken into account and its value depends on directivity of source beam, band-pass filter and non-imaging concentrator gain, but also the alignment of the emitter and the receiver and the distance between both. The path loss is inversely proportional to the square of the distance. For a non-LOS scenario the path loss is more complex to predict since it depends on multiple factors such as room dimensions, the reflectivity of the ceiling, walls and objects within the room, the position and orientation of the transmitter and receiver, window size and place and other physical matters within a room. Several models, mathematical and practical, are used to simulate some or all these factors. One of these models was carried out by Kahn and his team, who measure a maximum channel bandwidth of 300MHz. These tests were made for both scenarios (LOS and non-LOS) using different receiver locations in five different rooms and the results include impulse responses, path losses and RMS delay spreads.

One of the problems in VLC is the ambient light interference, created by the sunlight, incandescent lamps and fluorescent lamps. The sunlight radiation is typically the strongest source of noise, with a very wide spectral width and a maximum peak at $\sim 500nm$. When compared to the artificial light, sunlight can produce 1000 times more background current in the photodiode. This current acts like a DC current, that gives rise to shot noise independently of the signal. It is modelled as white noise. The artificial ambient light sources are modulated by the mains or, in some incandescent lamps, high frequency switching signal.

The radiation emitted by the incandescent lamps has an almost perfect 100Hz sinusoidal shape and has the maximum Power Spectral Density (PSD) at $1\mu m$, having the energy concentrated in the first few harmonics (up to 400Hz), thus, the high frequency component generates almost no noise. An effective way to mitigate noise from incandescent lamps is to use an high-pass filter. On the other hand, fluorescent lamps generates a distorted sinusoidal wave which can have frequencies up to 20KHz and can be filtered with a 4KHz high-pass filter without a significant system performance degradation. However, fluorescent lamps with electronic ballasts radiation, which works at 20-40KHz, generates noise from mains frequency and up to the megahertz range.

In outdoor VLC, the transmission channel behaves in a completely different way. In the one hand, the noise contribution factors are from a different nature than in indoor scenarios. On the other hand, those factors have a random component associated. There are a number

of statistical models to describe the outdoor channel characteristics.

The optical attenuation is modelled similarly to the indoor case, but the attenuation is higher due to the atmospheric conditions, such as fog, rain, clouds, and so forth. The attenuation can occur due to absorption and scattering. Absorption is wavelength selective and occurs when a photon is absorbed by a molecule and its energy is released in heat form. This problem can be mitigated using transmission windows that are not affected by absorption effect. Scattering is a result of the angular redistribution of the photon, with or without wavelength modification. This phenomenon depends on the particles size in the atmosphere (fog, clouds, rain) and also on the wavelength of the photon, λ. It can be classified as: 1) Rayleigh scattering, if the particles are much smaller than λ; 2) Mie scattering, if the particles are the same order of λ; 3) explained by diffraction theory (geometric optics) if the particles are bigger than λ.

2.10.5 Receiver Physical Layer

The Physical Layer of the ACO-OFDM and DCO-OFDM receiver is shown in Figure 2.15 and Figure 2.16, respectively.

Analysing the emitter and receiver diagrams it is easy to observe that the operations in the receiver are complementary to the operations in the emitter. The received signal from optical front-end is converted to digital using an Analog-to-Digital Converter (ADC), properly filtered to avoid aliasing and ambient light interference.

The ACO-OFDM demodulation includes CP remover, FFT operation, ACO-OFDM demapping to extract only the non-null subcarriers, Hermitian Symmetry rejection to remove the higher subcarriers added in the emitter, Deframing to recover the data and pilot subcarriers and an equalization process. The equalization is made in the subcarriers based on the channel estimation calculated with the pilot subcarriers. The DCO-OFDM process is identical but does not have the Demapping block.

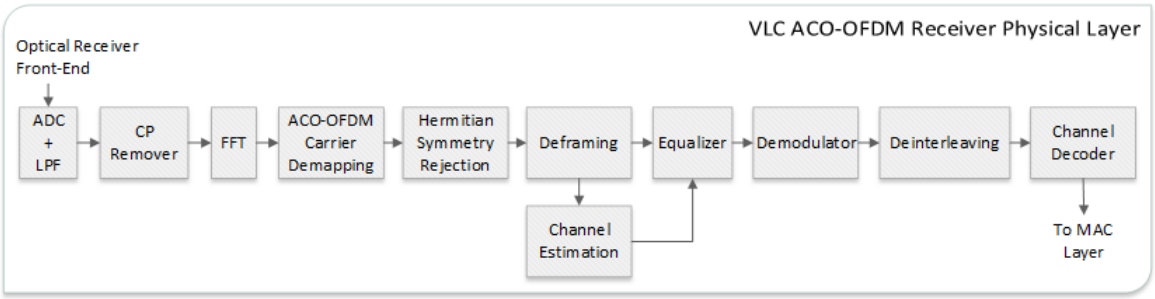

Figure 2.15: VLC ACO-OFDM Receiver Physical Layer

After the OFDM demodulation, the symbols goes through a Demodulator and then are processed in the FEC blocks: Deinterleaving and Channel Decoder. In this blocks the errors occurred during the transmission are detected and the ECC tries to correct the errors. The signal are then sent to the MAC Layer to recover the information from the MAC messages.

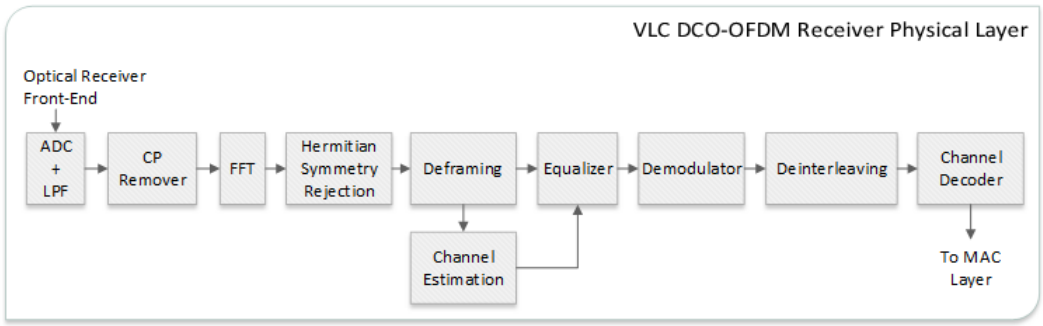

Figure 2.16: VLC DCO-OFDM Receiver Physical Layer

2.11 Tools and Devices for System Implementation

2.11.1 Spartan-6 FPGA SP605 Evaluation Kit

The models created in the previous section enables to simulate, with relatively small modifications, several configurations of different FEC techniques. This study provides useful information to decide which is the best solution to implement in the VLC system. However, these are only simulation models and can not be integrated in a real system. In order to implement FEC in the proposed system, the processing capabilities must be able to operate at dozens or even hundreds of MHz. In order to develop a real-time system having such characteristics the FEC techniques shall be implemented in hardware.

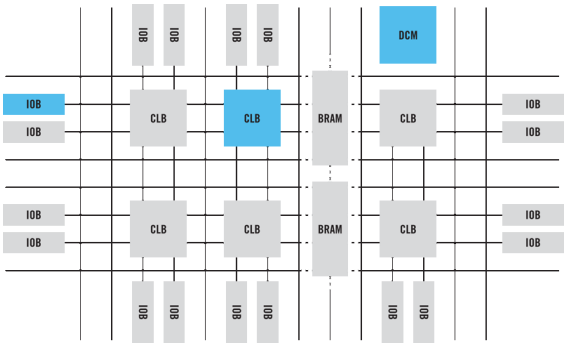

Figure 2.17: FPGA architecture [42]

A Field-Programmable Gate Array (FPGA) is a semiconductor device that can be programmed to a desired application after manufacturing. The principle of these devices are based on a matrix of Configurable Logic Blocks (CLBs) which can be interconnected. The CLBs common structure consists in lookup tables, flip-flops and multiplexers and can be configured to implement combinatorial logic, sift registers or memory. Modern FPGAs provide hard block that implement complex functionalities such as multipliers, RAMs, processors, among others. These blocks have in general configurable parameters which can be useful in many applications, providing high performance, simple integration, low power consumption

and the silicon area required. In order to acquire/retrieve data from input/output sources, FPGAs usually have IO Blocks which can be routed into the interconnect matrix. Other features such as RAM (Random Access Memory) and Clock Management are also implemented in the majority of actual devices.

The usage of FPGAs offers several advantages in the implementation process: Due to their architecture, these devices are highly configurable, providing design flexibility that gives to the developers the ability to add or remove features to the system; Comparing FPGAs to ASIC (Application Specific Integrated Circuits), the first ones offers faster and simpler prototyping, since it is only necessary to program the device in order to change system parameters while avoiding the required high-costs in ASIC development; On the other hand, comparing these devices to Micro-Controllers, FPGAs offers high speed signal processing and more complex systems can be implemented in a single chip.

Nonetheless, FPGAs development is not an optimized solution. Compared to ASIC, these devices require a larger silicon area and they have higher costs in large scale production. In addition, the FPGA architecture requires higher power consumption in contrast to ASIC and Micro-Controllers implementations.

Another disadvantage of FPGA system design is that the learning curve is higher due to the associated complexity since it requires hardware implementation in opposite to Micro-Controllers. The hardware design commonly uses HDL (Hardware Description Language) to describe the desired hardware architecture. In complex systems, the usage of HDL can be a challenging task. However, the FPGA companies usually have tools that can be used by developers in order to simplify the hardware design process.

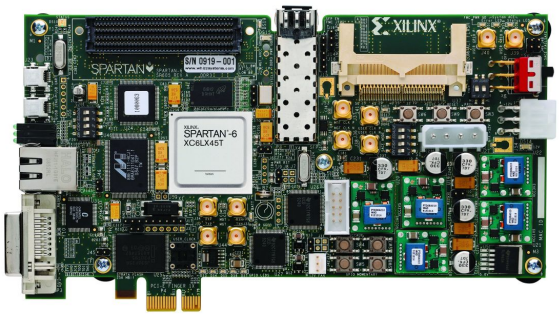

Figure 2.18: Spartan-6 FPGA SP605 Evaluation Kit

With such characteristics, the usage of an FPGA to implement the previous studied FEC techniques is the appropriate choice, providing both processing capabilities and design flexibility. This dissertation makes use of the *Spartan-6 FPGA SP605 Evaluation Kit* which is a development board featured with a *Spartan-6 XC6SLX45T-FGG484* FPGA, shown in Figure 2.18. The SP605 provides many features such as DDR3 memory, PCI Express interface, DVI, tri-mode Ethernet PHY, general purpose I/O and a UART, providing also the possibility to add others external functionalities using the industry-standard FMC (FPGA Mezzanine

Card) connector. In the VLCLighting project, the FMC connector was used to connect the output of the modulation to the ODAC.

The tools provided by Xilinx includes design tools (Xilinx ISE design suite and Xilinx System Generator) and debug tools (Xilinx ChipScope Pro). Xilinx ISE design suite is a group of software tools such as *Xilinx ISE Project Navigator*, *Xilinx Plan Ahead (XPS)* and *Xilinx Doftware Development Kit (SDK)*. The first one enables the developers to design a schematic of the desired logic circuit, using simple blocks such as adders, flip-flops, counters, among others. The *XPS* is used to create embedded processed-based systems, specifying the parameters of each component (processor and peripherals) as well as the interconnections between them. The *SDK* is used to create the software to a previous created hardware embedded processed-based system. The Xilinx System Generator tool makes use of Simulink environment to build hardware models and converting them to the FPGA HDL. The debug tool Xilinx ChipScope Pro inserts logic analyzer, system analyzer, and virtual I/O low-profile software cores directly into the design, providing the developers to analyse the internal signals in the FPGA.

2.11.2 Xilinx System Generator

The Xilinx System Generator integrates MatLab Simulink model design with Xilinx FPGAs. It offers to developers the possibility to design the system in Simulink environment using the blocks provided by Xilinx. At the end, the entire model can be synthesized without the user concern about the details of CLBs configurations, interconnections and clocking. Building a system model in System Generator also enables the possibility of its simulation in Simulink environment giving the developers the opportunity to analyse the system behaviour. Another interesting feature supported by this tool is called *hardware co-simulation* that combines the simulation environment from System Generator and the FPGA hardware resources to achieve faster speed in simulation process. Furthermore, a ChipScope Pro block can be inserted in the model. This tool is a logic analyser an can be used in the model debug. The input data of this block is obtained using the ChipScope Pro Analyser software provided by Xilinx.

2.11.3 Asynchronous Architecture

One of the most important VLCLighting characteristic is its Asynchronous architecture, which is shown in Figure 2.19.

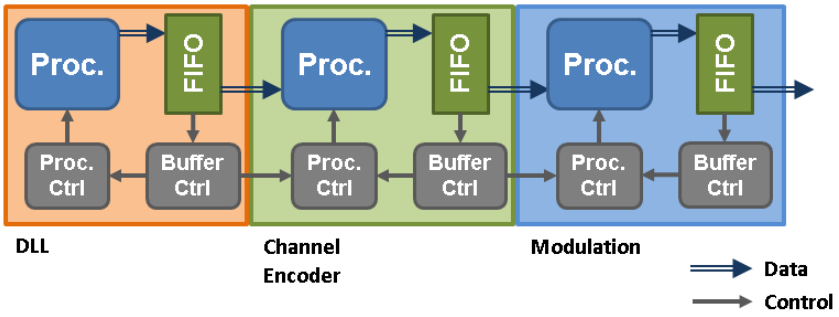

Figure 2.19: Asynchronous architecture - Emitter

The idea behind this implementation is to interconnect different blocks without the need of clock synchronization between functional blocks. Each block has a processing unit (Proc.) and an output buffer (FIFO). Besides, a buffer control block (Buffer Ctrl) analyse the fill state of the buffer and a process control block (Proc. Ctrl) starts or stops the processing unit, based on the buffers state. These control blocks are Finite State Machines (FSM).

The main block of asynchronous architecture is described in more detail in Figure 2.20.

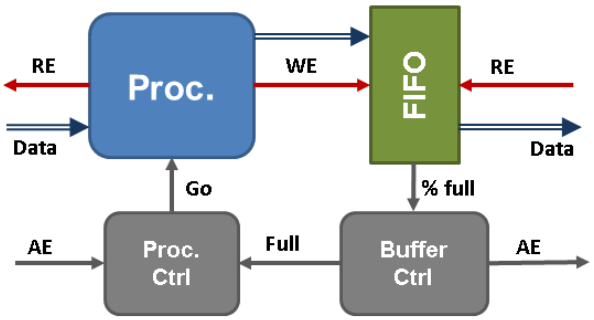

Figure 2.20: Asynchronous architecture emitter main block

The processing unit runs its operation until one of the two events occur: the FIFO in the current block is full or the FIFO of previous block is almost empty (AE). Each block sends its buffer status to the following block. In the Figure 2.21 is shown the structure of asynchronous architecture for the receiver side which is identical to the emitter.

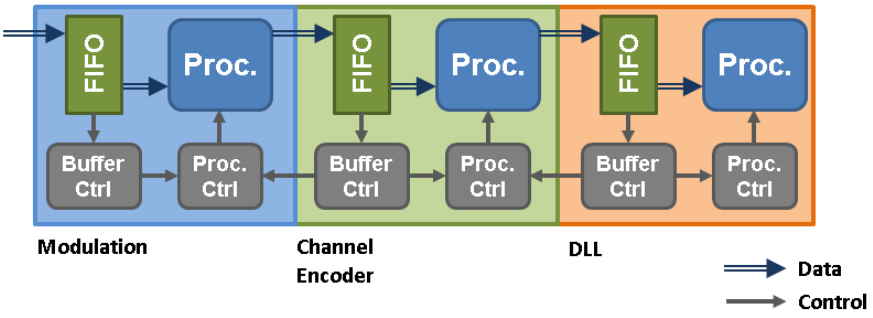

Figure 2.21: Asynchronous architecture - Receiver

As opposed to the emitter, each block has a buffer at the input The details of the receiver asynchronous block is shown in Figure 2.22.

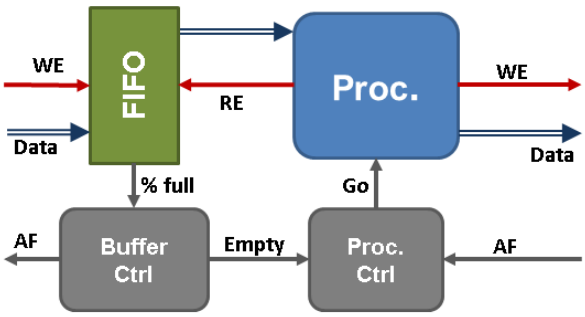

Figure 2.22: Asynchronous architecture receiver main block

Note that the receiver main block receives new data until the buffer is full and stops the processing if there are no more data in the FIFO or if the following block has its buffer is almost full (AF).

Using this architecture the blocks can be individual designed and tested since it is not restricted to the clock of the system. Furthermore, it is easy to add or remove blocks to a design without concerning the time constraints. Due to the design of each block does not take into account the clock speed, each block can be reused in systems with different speeds. However, this architecture has an extra complexity in each block, since it adds to the processing unit a buffer and two FSM. The extra components increases the hardware resource usage, particularly the FIFOs. Besides, the system presents an initial delay while the FIFOs are being fill.

2.12 Concluding Remarks

Through this chapter several VLC aspects, since the history to the actual technologies and recent achievements, were discussed. An overview of the FEC techniques and the system architecture was also described. It is clear that VLC research for high-speed communications is taking its first steps and there are many work to develop in this field.

Chapter 3

Channel Coding

The main goal of a well-designed communication system is to deliver the information from the emitter to the receiver without any difference from the original. Due to many reasons such as imperfections on modulation or detection algorithms, channel noise and interference, signal attenuation or bad signal coverage, the received data is affected by errors, leading the engineers to find solutions to improve systems performance.

In order to reduce transmission errors, several system parameters must be chosen carefully: signal bandwidth must meet the channel available bandwidth, modulation scheme should be selected according to the transmission medium and channel characteristics, noise must be filtered in the receiver, channel equalization, carrier and symbol synchronization, among others. Even after these improvements, there are some errors remaining in the transmission.

It is evident that, to have a solid communication system, some other methods must be used. Automatic Repeat Request (ARQ) is one of the techniques commonly used to control errors. This method works by sending acknowledgements for each correctly received data packet, resending it in case of the acknowledgement is not received or a timeout occurs without a required acknowledgement. In this way, if any error occur in the transmission, the data can be recovered in the retransmitted packet. However, ARQ needs a bidirectional transmission which is not the case of VLCLighting architecture.

An alternative method that can be useful in unidirectional communication systems adds redundant data to the transmission. Thus, in the receiver, this additional information is used to detect and correct some of the errors. These techniques are usually known as FEC and their performance depends on the used method and the error pattern.

3.1 VLC Channel Noise Sources

Considering that all the components of a VLC system were optimized (modulation, equalization, symbol and carrier synchronization, etc), most of the noise present in the received data is induced by the transmission channel. Thus, it is very important to study the noise sources in order to better understand what types of errors could be present and hence choose the proper FEC technique(s) to be used in the system.

As explained previously in section 2.10.4, many noise sources affects the system performance. For public lighting VLC applications the distances between emitter and receiver can vary from 1m to 10m and both LOS and non-LOS scenarios must be taken into account.

In the LOS scenario, the received light power varies according to the arrangement of the

emitter and the receiver. As this system uses LEDs which intensity radiation is modelled by a Lambertian distribution, shown in equation 3.1.

$$R_0(\Phi) = \begin{cases} \frac{m_1+1}{2\pi} cos^{m_1}(\Phi) & \text{for } \Phi \in [-\pi/2, \pi/2] \\ 0 & \text{for } \Phi \geq \pi/2 \end{cases} \tag{3.1}$$

where m_1 is the Lamberts mode number expressing directivity of the source beam, related to the LED semiangle at half-power $\Phi_{1/2}$ by

$$m_1 = \frac{-ln\, 2}{ln(cos\ \Phi_{1/2})} \tag{3.2}$$

The total loss is given by the equation 3.3.

$$H_{los}(0) = \begin{cases} \frac{A_r m_1+1}{2\pi d^2} cos^{m_1}(\Phi) T_s(\psi) g(\psi) cos\psi & \text{for } 0 \leq \psi \leq \Psi_c \\ 0 & \text{elsewhere} \end{cases} \tag{3.3}$$

where A_r is the active area of the receiver, ψ is the incident angle in the receiver, $T_s(\psi)$ is the transmission of the optical band-pass filter, $g(\psi)$ is the gain of the non-imaging concentrator, d is the distance between emitter and receiver and Φ is the transmitting angle. Ψ is the Field of View (FOV) of the detector. Figure 3.1 resumes the LOS propagation model.

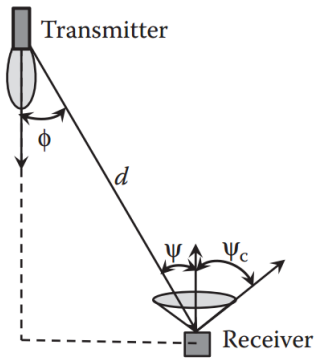

Figure 3.1: LOS propagation model

The received optical power P_r, in LOS is obtained by applying the gain in 3.3 to the transmitted power P_t.

$$P_r = H_{los}(0) P_t \tag{3.4}$$

Some scenarios, such LOS blockage, requires a non-LOS scheme. In such conditions, the receiver must rely on reflections in the surrounding environment to receive the signal. The path loss is considerably higher than in the previous case due to a greater distance from the emitter and also due to the light absorption by the reflecting surfaces. Note that the received signal is composed by multiple reflections with similar Power Spectral Density and different delays which can cause ISI, decreasing system performance. Despite in LOS propagation these reflections have lower magnitude when compared to the direct signal, there will be also some (less) negative impact in the system performance.

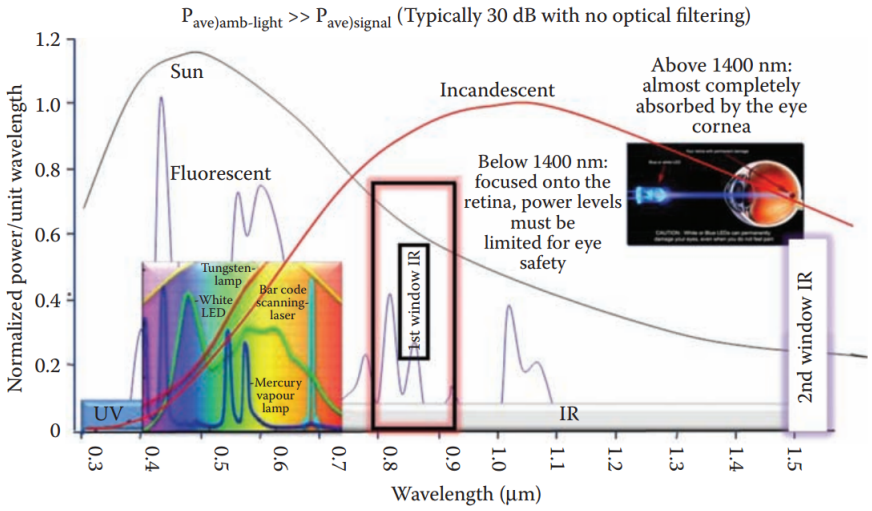

Figure 3.2: Normalized power/unit wavelength for optical wireless spectrum and ambient light sources [4]

International Visibility Code					
Atmospheric Conditions	**Weather Constituents (mm/h)**			**Visibility (m)**	**Attenuation (dB/km)**
Dense fog				50	315
Thick fog				200	75
Moderate fog				500	28.9
Light fog	Snow	Storm	100	770	18.3
Very light fog				1000	13.8
		Strong rain	25	1900	6.9
Light mist				2000	6.6
		Average rain	12.5	2800	4.6
Very light mist				4000	3.1
		Light rain	2.5	5900	2
				10,000	1.1
Clear air		Drizzle	0.25	18,100	0.6
				20,000	0.54
Very clear air				23,000	0.47
				50,000	0.19

Table 3.1: International Visibility range and Attenuation coefficient in the Visible Waveband for Various Weather conditions

Usually, in VLC outdoor systems most of the noise comes from ambient light, predominantly from the sun, as shown in Figure 3.2, but also from other light sources such as vehicles headlights, existing public lighting systems or moonlight. The average combined power of the ambient light induces a DC background current that giving rise to shot noise which is independent of the signal and can be modelled as an AWGN.

In addition to ambient light, the channel atmospheric attenuation also causes signal degradation. Most common factors to atmospheric attenuation are fog, rain, clouds and snow. Table 3.1 presents some visibility ranges and attenuation coefficients for several atmospheric conditions.

For distances up to 10 meters, only the dense, thick and moderate fog scenarios can cause a significant signal attenuation, around 3.15dB, 0.75dB and 0.29dB, respectively. The remaining conditions attenuates the received optical power in less than 5%. However, it must be noted that the number of errors in the transmission increases, even for lower attenuation scenarios.

3.2 Error patterns

With simplicity in mind, the blocks from Modulator on, in section 2.10.3, were considered as a Binary Symmetric Channel (BSC). This channel transmits binary symbols and the errors that occurs in the channel have the same probability independently of the transmitted symbol (Symmetric).

Therefore, errors induced by previously mentioned noise sources can be classified in two groups: random errors and burst errors.

Random Errors can occur unpredictably in the channel, with a probability of error P_e. The common source of random errors is AWGN. A model commonly used to simulate the generation of random errors is the well known Bernoulli Distribution (equation 3.5).

$$f(k;p) = p^k(1-p)^{(1-k)} \quad \text{for } k \in \{0,1\} \tag{3.5}$$

On the other hand, Burst Errors are characterized by a sequence of errors with a certain length. There are several models that can describe this process. One of the commonly used is called Gilbert-Elliott Model. The model considers two channel states: a good state (G), with an error probability of $1-k$ and a bad state (B) having an error probability of $1-h$. Furthermore, the channel changes its state from good to bad with a probability of p and in the opposite way with a probability of r. A visual representation of this model is seen in Figure 3.3 [43].

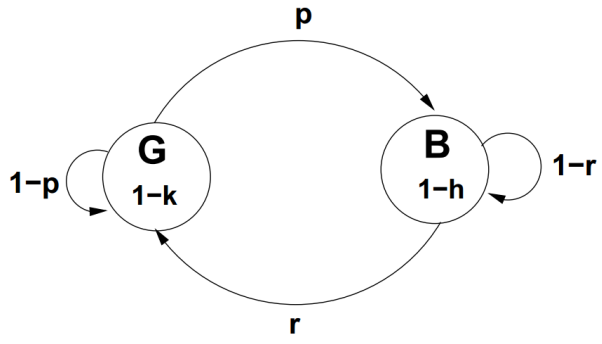

Figure 3.3: Gilbert-Elliott Model with two states

The stationary state probabilities π_G and π_B exist for $0 < p$, $r < 1$.

$$\pi_G = \frac{r}{p+r}, \quad \pi_B = \frac{p}{p+r} \tag{3.6}$$

Given (3.6), the error probability resulting from this model is

$$P_e = (1-k)\pi_G + (1-h)\pi_B \tag{3.7}$$

3.3 Bit Error Rate analysis

This section will present the BER estimation for communication systems using QPSK modulation over an AWGN channel.

For this scheme, the errors at the output of the QPSK Demodulator should present a random pattern. A closer look to VLCLighting architecture should be made in order to explain this approach. The effect of OFDM modulation can be understood as a group of multiple QPSK symbols since each subcarrier modulates one symbol. In addition, each subcarrier should present the same SNR due to the usage of channel estimation and equalisation. Moreover, consecutive symbols are mapped in different subcarriers affected by different types of noise. The output of the Demodulator should present a BER that follows closely the curve of the AWGN channel model. The Interleaving block is a reinforcement of this approach, changing the remaining burst errors to random errors.

The SNR is a relation between the signal power, P_S, and noise power, P_N, usually expressed in decibels [44].

$$SNR(dB) = 10log_{10}\left(\frac{P_S}{P_N}\right) \tag{3.8}$$

In digital communications is commonly used a relation between the energy per bit, E_b, and the noise power spectral density, N_0 [44]:

$$\frac{E_b}{N_0} = \frac{W}{R} \times \frac{P_S}{P_N} = \frac{P_S}{\rho P_N} \tag{3.9}$$

where W is the channel bandwidth, R is bit rate and ρ is the spectral efficiency. The error probability of QPSK modulation through an AWGN channel is given by:

$$P_e = Q\left(\sqrt{2\frac{E_b}{N_0}}\right) \tag{3.10}$$

Note that the error probability depends exclusively on E_b/N_0. The function Q in expression 3.10 can be calculated using the formula 3.11

$$Q(x) = \frac{1}{2}\, erfc\left(\frac{x}{\sqrt{2}}\right) = \frac{1}{2}\mathrm{x}\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2} dt \tag{3.11}$$

3.4 Reed-Solomon Codes

Widely used in current communication and storage systems such as hard disks, CD, DVD, barcodes, DVB, xDSL and deep space and satellite communications [30], Reed-Solomon Codes are one of the many techniques to control undesirable transmission errors.

The first description of these codes was in June 1960, in a paper titled "Polynomial Codes over Certain Finite Fields" by Reed and Solomon. Later, it was proved that Reed-Solomon codes and BCH codes are related and, in fact, they are non-binary BCH codes.

The RS codes are constructed over Galois Field (finite field). These fields have a finite number of elements. Besides, the operations between the elements always result in an element of the field.

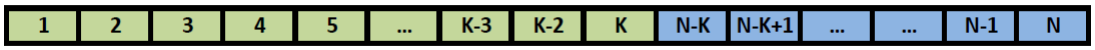

Figure 3.4: RS(N,K) codeword structure

A RS(N,K) code have length N and dimension K over a GF(q), where $q = p^m$, $p = 2$. This code has minimum distance $d = N - K + 1$ and N=q-1. In other words, is a code that has N symbols, being K symbols formed by the data to be encoded. Each symbol is formed by $m = log_2(N + 1)$ bits. Is easy to observe that, for the actual majority of communication and storage systems, the RS(255,K) is the common choice, since each symbol has 8bits, making these set of codes ideal to be integrated in systems where information is transmitted in Bytes.

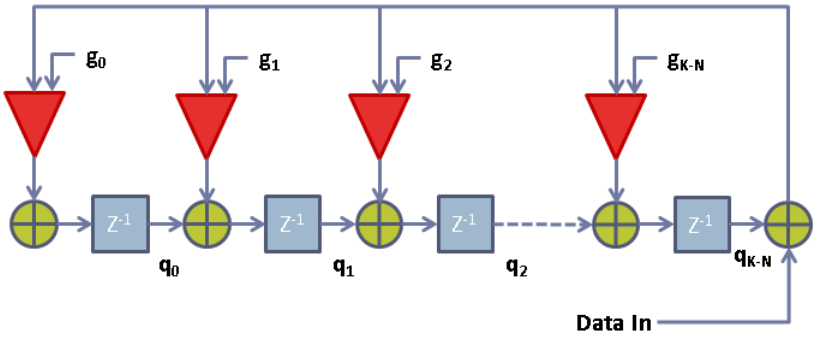

Figure 3.5: Reed-Solomon encoder hardware implementation

The codeword is composed by two distinct parts, shown in Figure 3.4: the first one is simply the data to be encoded, divided in K symbols. The second one has the redundant $N - K$ symbols to be used later in the decoding process, known as check symbols or parity symbols.

The RS encoder works by dividing the information (in polynomial form) to be encoded by the generator polynomial. The result of this operation is directly the check symbols. Figure 3.5 represents a hardware implementation of a RS encoder, using the polynomial division algorithm. The multiplier coefficients g_n are the coefficients of the RS generator polynomial.

The decoding process, shown in Figure 3.6, is more complex. At the receiver, the syndromes for each codeword are calculated to determine the number of errors. The error locations and error values are computed and using this information an error pattern is obtained and subtracted to the received codeword, resulting in the most probable original sent codeword.

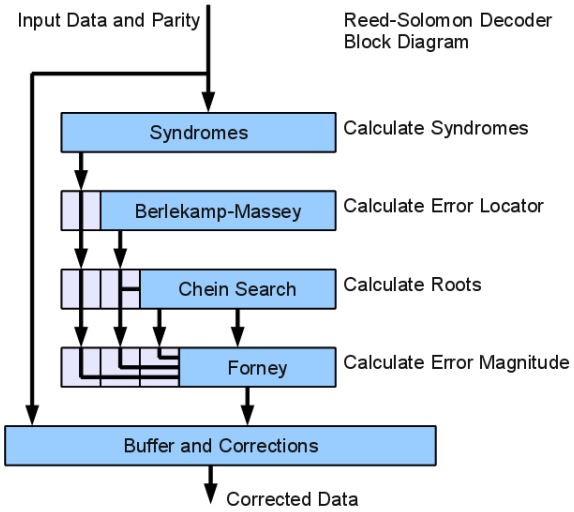

Figure 3.6: Reed-Solomon decoder diagram block [45]

To use RS codes in practical applications some of their parameters must be taken into account. In fact, the only variables are N and K, which determines the behaviour of the code.

Given a RS(N,K) code, the code rate, R, is the proportion between the data symbols and the total symbols in a codeword, which also represents the coding efficiency.

$$R = \frac{K}{N} \tag{3.12}$$

RS codes are known for their good distance proprieties, satisfy the singleton bound (3.13) with equality.

$$K + D_{min} \leq N + 1 \tag{3.13}$$

The guaranteed correction capability is expressed in (3.14), where $D_{min} = N - K + 1$ is the minimum distance of the code.

$$t = \frac{1}{2}(D_{min} - 1) = \frac{1}{2}(N - K) \tag{3.14}$$

3.4.1 BER Analytical Estimation for Reed-Solomon Codes

In order to study the performance in terms of the decoded codeword error probability, it will be used a symmetric memoryless channel, shown in Figure 3.7, which is a generalization of the BSC previously mentioned. In this channel model, the output symbol is not affected

by the input symbol or by the previous symbols. Taking this into account, the RS codeword error probability is upper bounded by:

$$P_e \leq \sum_{i=t+1}^{N} \binom{N}{i} P_M^i (1 - P_M^i)^{N-i} \tag{3.15}$$

where $1 - P_M$ represents the probability of a symbol transmission without any errors. The corresponding symbol error probability is

$$P_{es} = \frac{1}{N} \sum_{i=t+1}^{N} i \binom{N}{i} P_M^i (1 - P_M^i)^{N-i} \tag{3.16}$$

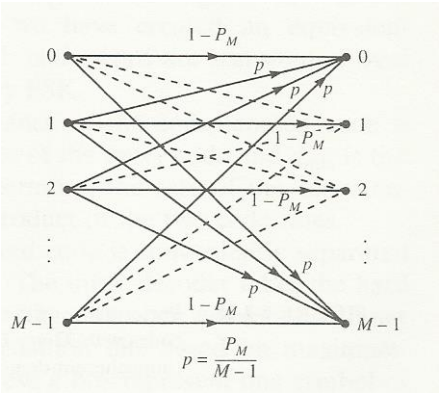

Figure 3.7: M-ary input, M-ary output, symmetric memoryless channel [46]

From equation 3.16, bit error probability is

$$P_{eb} = \frac{2^{K-1}}{2^K - 1} P_{es} \tag{3.17}$$

Considering a low channel error probability, an approximation in the bit error probability should be made, resulting in $P_{eb} \approx P_{es}$, since it is unlikely to have more than one bit error per symbol.

The following result assumes a system using QPSK and an AWGN channel. For a particular RS(N,K) code, P_M is given by:

$$P_M = 1 - \left(1 - Q\left(\sqrt{2R\frac{E_b}{N_0}}\right)\right)^m \tag{3.18}$$

3.5 Convolutional Codes

Convolutional codes are different from block codes since they do not have a strict codeword. The encoding method uses a linear finite-state shift register with length K. The input data, in bits, is shifted in the shift register k bits at a time. The encoded information is a sequence if n bits, which are the result of n modulo 2 additions. The parameter K is called the constraint length and it has impact on the code performance. As in the block codes, the

code rate is given by $R = K/n$ and it represents how many bits the code outputs for each input bit. The Figure 3.8 shows a generic convolutional encoder.

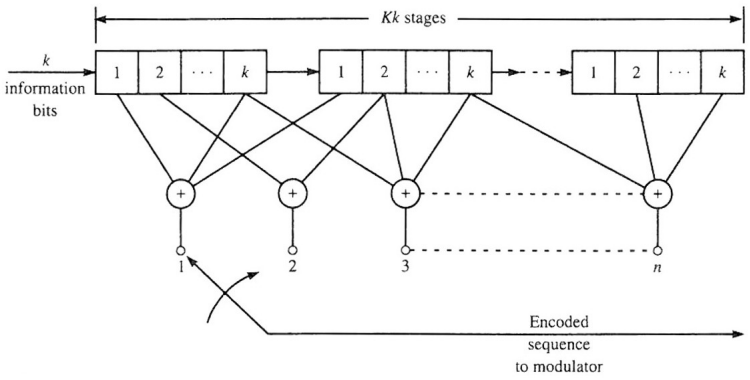

Figure 3.8: Convolutional Encoder [47]

There are different approaches to describe a convolutional code. One method is to establish the function generators for each output bit. For instance, lets consider a convolutional code with a constraint length $K = 3$ and $R = 1/2$, which mean it has 2 outputs, as seen in Figure 3.9. Each output is described by a function generator vector.

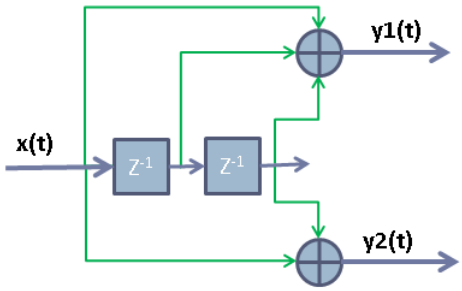

Figure 3.9: Convolutional Encoder K=3, n=2, k=1

For this particular code, the generator vectors are:

$$g_1 = [111] \tag{3.19}$$

$$g_2 = [101] \tag{3.20}$$

For convenience, generators are usually represented in octal notation, which gives $g_1 = 7_8$ and $g_2 = 5_8$.

Alternative approaches are the Trellis diagram and the state machine.

The state diagram describes how the output changes, considering the actual state and the new input bit, as shown in Figure 3.10. The construction of this diagram starts from the zero state (all outputs have zero value) and develop the transitions between each state.

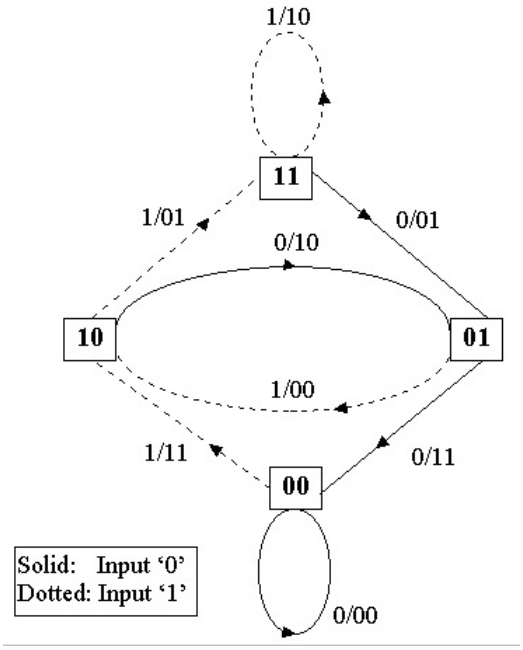

Figure 3.10: State machine of a convolutional code K=3, n=2, k=1 [48]

The analysis of a state machine can be difficult since it could go through the same state several times, loosing the information about previous visited states. A more detailed view for state diagrams is the Trellis diagram. This diagrams expands state machines in time giving the possibility to view how the states transitions evolves in time.

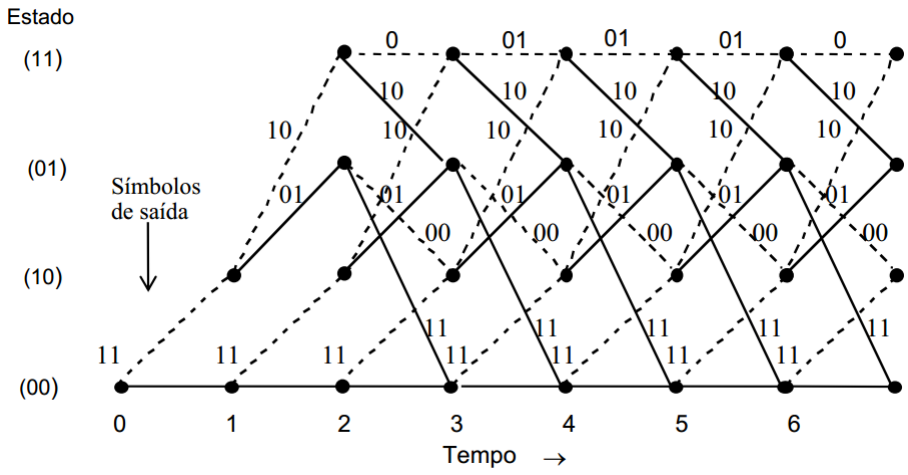

Figure 3.11: Trellis diagram of a convolutional code K=3, n=2, k=1 [48]

The commonly used decoding scheme uses a Viterbi decoder which computes the minimum Hamming distance between the received codeword and all the possible codewords to find the most probable one. The method used is called trace-back decoding which stores all the

previous states and transitions. Branch metrics are used in order to determine which are the most probable path, thus retrieving the most probable sequence of transmitted information. Hamming distance is commonly used to compute the most likely transitions in each branch of the Trellis structure.

Unlike block codes, convolutional codes does not have a fixed codeword length since it may change depending on the input data. The Viterbi decoder sets the end of a codeword when, starting from the zero state, the decoder enters in that state again. Furthermore, if the zero state is not reached in a predefined number of transitions, the decoder starts to analyse the received information. The number of maximum transitions is called the decoding depth.

The minimum distance free, d_{free} is given by:

$$d_{free} \leq min_{l \geq 1} \left\lfloor \frac{2^{l-1}}{2^l - 1}(K + l - 1)n \right\rfloor \tag{3.21}$$

where $\lfloor x \rfloor$ denotes the largest integer contained in x. Table 3.2 shows the maximum free distance for several constraint lengths, indicating, for each case, the generators to achieve this.

Constraint Length K	Generators in Octal		d_{free}	Upper bound on d_{free}
3	5	7	5	5
4	15	17	6	6
5	23	35	7	8
6	53	75	8	8
7	133	171	10	10
8	247	371	10	11
9	561	753	12	12
10	1167	1545	12	13
11	2335	3661	14	14
12	4335	5723	15	15
13	10533	17661	16	16

Table 3.2: Rate 1/2 maximum free distance code

3.5.1 BER Analytical Estimation for Convolutional Codes

The performance of convolutional codes for a BSC is approximated by:

$$P_b \approx b_{d_{free}} \left(2\sqrt{p(1-p)}\right)^{d_{free}} \tag{3.22}$$

where p is the probability of a transition in the channel, d_{free} is the free distance of the code and $b_{d_{free}}$ is the number of nonzero information bits associated with codewords of weight d_{free}.

A more practical approach to study the performance of convolutional codes is to estimate the coding gain over AWGN channels. This measure represents the difference between the E_b/N_0 of the uncoded and the coded system at a particular error probability. The coding gain for a particular code depends on the code rate and on the free distance of the code. For

a soft-decision decoder, the upper bound on the coding gain over an uncoded binary QPSK system is:

$$C_G \leq 10log_{10}(R\, d_{free}) \tag{3.23}$$

	Eb/No uncoded	$R_c = 1/3$		$R_c = 1/2$		
P_b	(dB)	K=7	K=8	K=5	K=6	K=7
10^{-3}	6.8	4.2	4.4	3.3	3.5	3.8
10^{-5}	9.6	5.7	5.9	4.3	4.6	5.1
10^{-7}	11.3	6.2	6.5	4.9	5.3	5.5

Table 3.3: Coding gain for soft-decision Viterbi decoder

In the Table 3.3, are shown coding gain values for several codes with different code rates and constraint lengths. Note that these values are for soft-decision decoder. The hard-decision decoder coding gains are reduced by approximately 2dB. The difference between hard and soft decision relies on the comparison method used to identify the most probable symbol, giving an received symbol. In the first case the received symbol is compared to all possible symbols and the most probable symbol is the one which has lower Hamming distance. The second method uses Euclidean distance to compare the received symbol with all possible symbols.

3.6 CRC

Even with the usage of ECC some errors may remain in the decoded data. Thus, after the decoding process, it is essential to evaluate the integrity of information. One way to do this is by introducing a single parity check bit, which is the modulo 2 addition of all transmitted bits. In this method, the parity bit is computed at the emitter side and in the receiver, the transmitted parity bit is compared to a new one, computed with the received information. It is evident that the parity bit technique can only detect errors, because it is impossible to know where the error is in order to correct it. Is also important to note that this technique can only detect an odd number of errors.

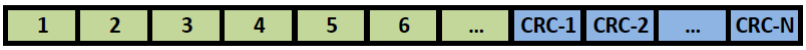

Figure 3.12: CRC-N codeword

A more sophisticated method is called Cyclic Redundancy Check (CRC) and it is based on systematic cyclic codes which are well-known for their burst detection capability. This method is equivalent to the parity bit technique, however, instead of a simple modulo 2 addition of all transmitted bits, the information in divided by a polynomial and the reminder is appended to the transmitted data, as shown in Figure 3.12. This operation is made according the finite field rules. Equation 3.24 shows the polynomial division operation where $M(x)$ is the message to be sent, $G(x)$ is the generator polynomial with N terms and $R(x)$ is the remainder

polynomial.

$$R(x) = M(x)x^N \, mod \, G(x) \tag{3.24}$$

At the receiver the same operation is made, with the same polynomial, and the reminder must be equal to the CRC sent. Other method divides both information and CRC field and the result of the remainder must be zero, if no errors occurred in the transmission.

A CRC that has N bits, also known as CRC-N, is generated by a polynomial that has $N+1$ terms. The maximum bits that can be encoded by a CRC-N is $(2^{N-1}-1)-N$. This code can detect all burst error of length N or less and furthermore, it detects all burst errors of lengths above $N+1$ with a probability of $1-2^{-N}$. The worst detection capability occurs in the scenario of a burst error of length $N+1$ where the probability of detection is given by:

$$P_{detection} = 1 - 2^{1-N} \tag{3.25}$$

3.7 Interleaving

The majority of ECC improves the communication system performance for random errors. However, some channels present burst errors as in the case of multipath scenario. The RS and Convolutional codes, in particular the Convolutional codes with lower constraint lengths, are designed for random error scenarios. However, they can correct some burst errors. An upper bound for burst correction for both codes is given by:

$$b \leq \frac{1}{2}(N-K) \tag{3.26}$$

However, in general, burst errors are beyond these codes correction capability. To mitigate the effect of burst errors in communication systems, it is commonly used a technique that transforms burst errors into random errors. This method is called Interleaving and is widely used together with codes that presents high performance in random errors.

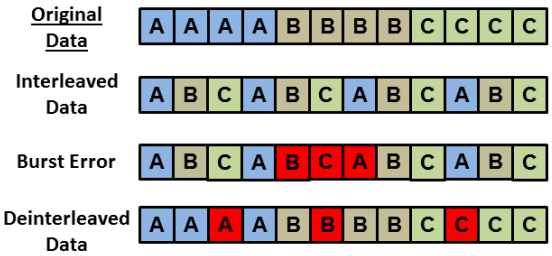

Figure 3.13: Interleaving example

The principle behind Interleaving is simple. At the emitter, the interleaver changes the order of the transmitted symbols, in a known pattern. When a burst error is added to the transmission, the sequence of affected symbols are different from the original sequence. At the receiver, the received symbols are deinterleaved, restoring the original sequence of information. In this step, the burst error was spread along the received data while the symbols are in the correct order. Figure 3.13 helps to visualise the Interleaving process and how it may reduce

burst errors. Lets assume that the codewords A, B and C are block codes formed by 4 symbols and the corection capability of these codes are 1 symbol. Even after a burst error affecting 3 symbols, all codewords can be recovered at the receiver. There are two main Interleaving techniques: block interleaving and convolutional interleaving.

The block interleaver is commonly used for interleaving block codes and it is a matrix with N columns and m rows. The number of columns is chosen to match the number of symbols in a block code. Therefore, each row stores a complete codeword and the matrix is capable of storing m codewords. While the codewords are read into the matrix row-wise, the output information is read column-wise, as shown in Figure 3.14.

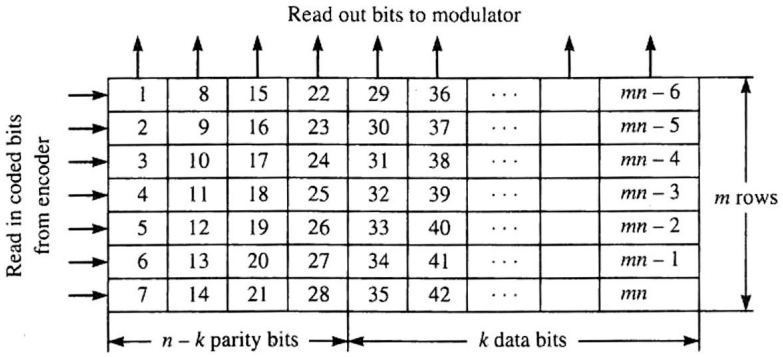

Figure 3.14: Block interleaver matrix [47]

The output result is identical to Figure 3.13, assuming $N = 4$ and $m = 3$. A burst error of length b causes a maximum of b/m errors to occur in one or more codewords. Considering a block code that can correct up to t symbols, using a block interleaver the total correction capability is improved to mt symbols.

It is important to note that block interleavers adds a delay comparatively to a system without interleavers. The latency is related to the matrix dimensions and its value is $2(m \times N)t_s$, which is the time to read out the matrix at the emitter side and to read in at the receiver side, where t_s is the duration of one symbol.

The Convolutional Interleaver uses $B-1$ shift registers with different fixed delays as shown in Figure 3.15.

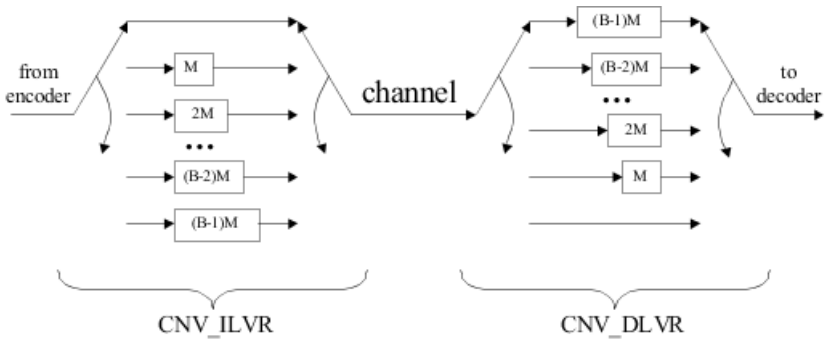

Figure 3.15: Convolutional interleaver and deinterleaver [49]

An example of the Convolutional Interleaver output with 3 stages is represented in Figure 3.16.

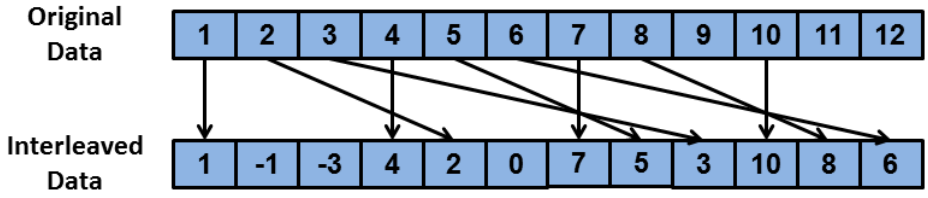

Figure 3.16: Convolutional interleaving output

Since the maximum delay block has latency $(B-1)t_s$ and each input symbol must wait B branches to leave the shift register, the total latency introduced by this method is given by $B(B-1)t_s$. This scheme is commonly used with Convolutional codes.

3.8 Puncturing

Puncturing is a technique commonly used in communication systems to increase the transmission rate. The main idea behind puncturing is to remove some information after the encoder process. At the receiver side, the decoder knows the positions of missing data and fill them with zeros. If the missing data is below the system error correction capability, assuming that the channel does not inserts errors above this limit, the ECC can correct all the errors introduced by puncturing.

This technique allows the usage of a fixed ECC which makes the implementation process easier. However the amount of punctured information must be controlled in order to accommodate possible errors in the channel. In other words, if too many information is removed and the channel presents high error probability, the decoder may not be able to recover all the data. In order to employ this technique without compromising the system performance, an adequate level of puncturing should be applied based on the channel error probability status. In bi-directional systems, the receiver can inform the emitter about channel status. However, in a broadcast system, the emitter can not retrieve the channel status. Thus, using puncturing could lead to a system performance degradation and will not be considered in the Channel Encoder implementation for VLCLighting system.

3.9 Concluding Remarks

In this chapter were described the analytical tools to predict different FEC techniques behaviour. While convolutional codes have capability to correct random errors, RS codes can correct small burst errors of size $N-K$. The CRC seams an interesting feature to integrate in a system in order to provide data validation. Interleaving should avoid large burst errors by turning burst errors into random errors. Puncturing will not be considered in models design since the emitter has no capability of predict channel status.

Chapter 4

System Design Considerations

In the previous chapter were established the characteristics of different types of errors (random and busrt) as well as RS and Convolutional codes, CRC and Interleaving schemes. In order to proper understanding how FEC techniques behave in different scenarios several models will be discussed in this chapter.

To first study the various techniques without concerning the hardware implementation details, was used the Matlab Simulink tool to create simple models whose results will be compared to the theoretical expected values. Later, the models will be implemented in a Xilinx System Generator environment and both simulations and hardware co-simulations will be performed.

At the end of the chapter, the results will be compared to decide which FEC techniques combination should be considered for the Channel Encoder.

4.1 Simulink Models

Before going into detail on ECC models, first it shall be created a model that can describe a BSC. This model, represented in Figure 4.1, contains 3 main blocks: *Data Generator*, *Error Generator* and an *Additive Noise Channel*.

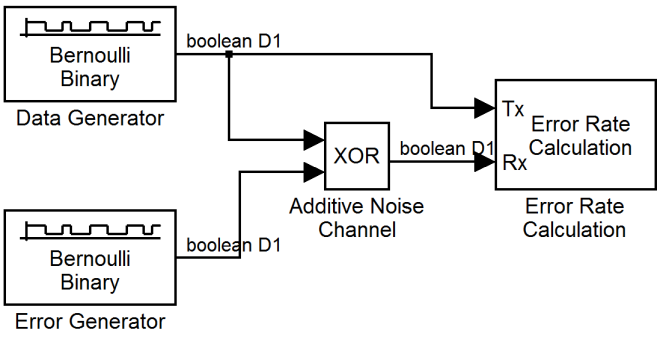

Figure 4.1: BSC simulink model

At the output of the *Data Generator* block, bits are generated, one at a time, based on a Bernoulli distribution with equal probability of zeros and ones. The block *Error Generator* produces errors (ones) with a probability of P_b. The *Additive Noise Channel* is realized by

an logical XOR operation of data bits and errors, resulting in a transition of the data bit when there is an error in the channel. The fourth block, *Error Rate Calculation*, compares the transmitted and received data and computes an error rate, which must be approximately P_b. The data and error generator blocks have a fixed sample time $D1$.

In Figure 4.2 it is shown the results for the BSC model. The simulated probability of error values are from 10^{-6} to 10^{-2}, with logarithmic steps, and for each value, 10^7 bits were simulated. As the reader can observe, the simulated probability of error curve approaches the expected values.

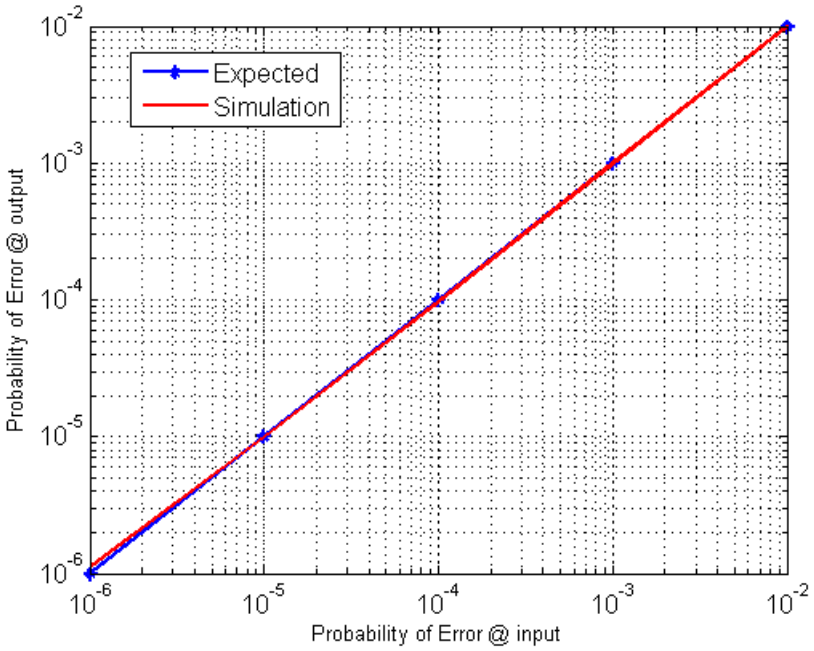

Figure 4.2: BSC Simulink model simluation results

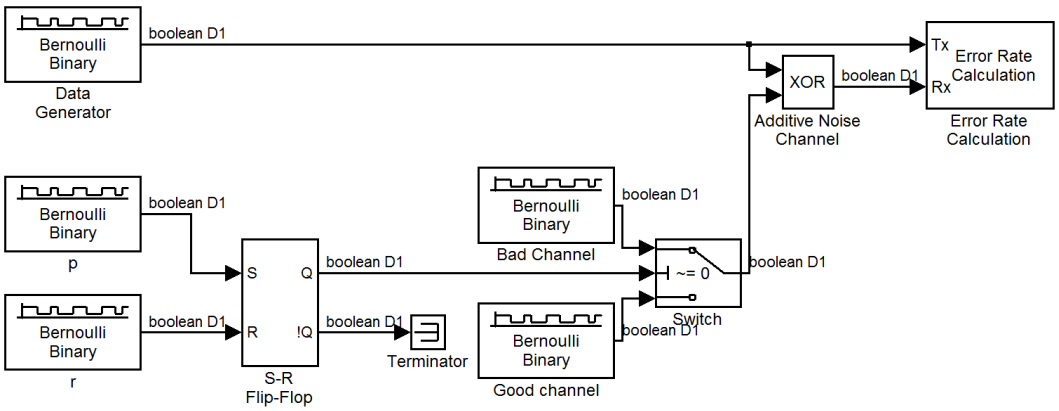

Figure 4.3: Gilbert-Elliott simulink model, for burst error simulation

The Gilbert-Elliott Simulink model used for burst error, previously characterized in Section 3.2, is shown in Figure 4.3. The blocks *Data Generator*, *Additive Noise Channel* and *Error Rate Calculation* remain the same from the BSC model, however the *Error Generator* was replaced by a state machine, as the model suggests. The model has two channel states with different error probabilities: the probability of error produced by *Bad Channel* and *Good Channel* blocks are $1-h$ and $1-k$, respectively. The *S-R Flip-Flop* switches the channel state by driving the "Switch" block according to the state transitions probabilities r and p.

The results for this model simulation with 10^7 bits are shown in Figure 4.4.

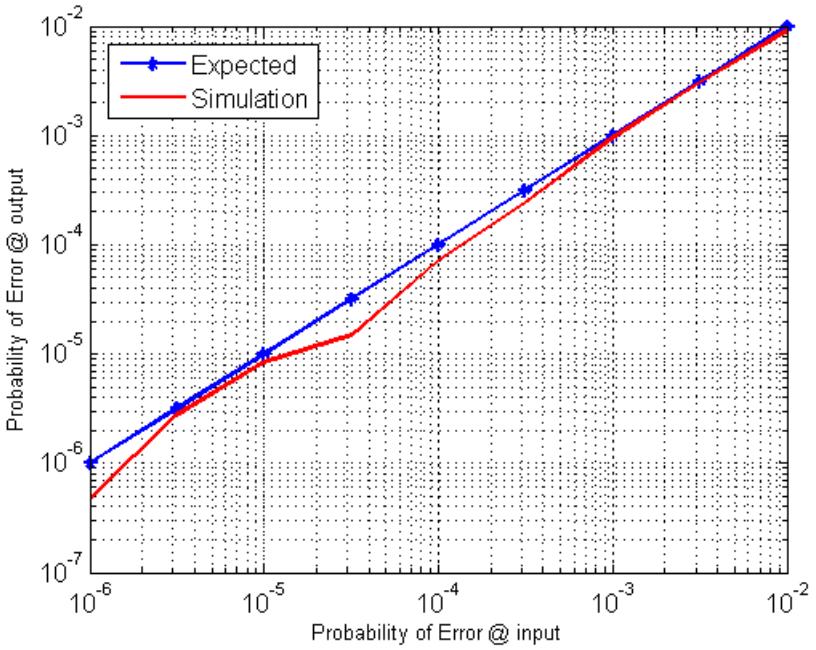

Figure 4.4: Gilbert-Elliott Simulink model simluation results

The parameter h of Gilbert-Elliott model is equal to zero, in order to provide a continuous burst error. Since r represents the probability of a transition from bad state to good state, the mean value of the burst length is given by $1/r$. The probability of error in the good state, $1-k$, was considered to be the desired error probability divided by the length of burst errors. The last parameter, p, is computed taking into account equations 3.6 and 3.7, resulting in:

$$p = \frac{r(1-k-P_e)}{P_e - 1} \tag{4.1}$$

The simulation result follows the expected error probability. Besides the output error probability, it is also interesting to analyse the types of burst errors generated by the Gilbert-Elliott model. The script used to make the statistical analysis of burst errors can be seen in Appendix D and it is a state machine that analyses the sequence of bits generated by the Gilbert-Elliott model. It only considers the burst errors and it computes each burst error length in a vector.

The result of this estimation for an input error probability of 10^{-3} is listed below:

- Number of burst errors: 83
- Average burst length: 113 bits
- Maximum burst length: 875 bits
- Minimum burst length: 2 bits

As expected, the average burst error length is approximately 100 bits and can be any size around this value.

Once the channel models were established we shall now proceed to the ECC models. In Figure 4.5 is represented the RS(255,213) simulink model in presence of random errors. This model is based on the BSC simulink model where it was added a *RS encoder* and a *RS decoder*. For model analysis a *Channel Error Rate* block was included in order to verify if the simulated channel error rate matches the desired value.

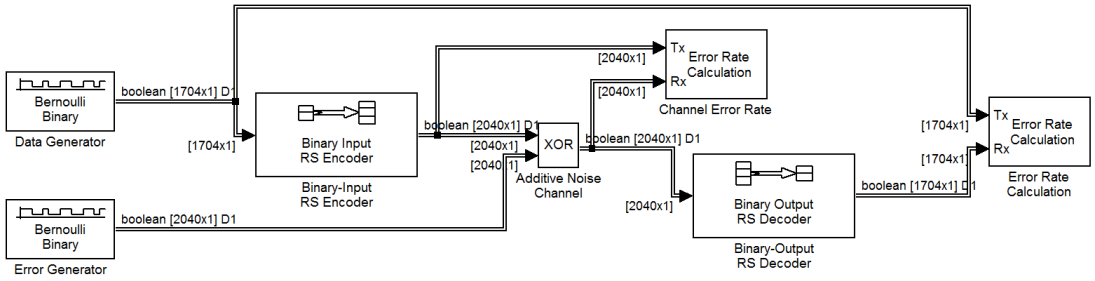

Figure 4.5: Reed-Solomon simulink model in presence of random errors

Several modifications in the data and generator blocks were made in order to synchronize the system and provide a frame-based simulation, which is required by RS encoder and decoder blocks and it also improves simulation performance. The sample time in the *Data Generator* remains the same but the output is now a frame of $K \times m$ bits, where K is the number of data symbols in the Reed-Solomon code and m is the number of bits of each symbol. Thus, a frame will be generated every $t_s(K \times m)$, where t_s is the sample time.

It is very important to note that, at the output of the *RS encoder* the number of bits is higher that in the input. In fact, the output frame will have $N \times m$ bits, for the same period of time. Therefore, the bit generation sample time in the *Error Generator* must be decreased by a factor of R, the code rate. In addition, the output frame size must match to the output of the *RS encoder* which is $N \times m$. These configurations grants one complete error frame for each data frame.

In the Reed-Solomon encoder and decoder blocks the code parameters must be declared:

- Codeword Length N = N (variable to be defined in MatLab script)
- Message Length K = K (variable to be defined in MatLab script)
- Specify primitive polynomial = unchecked

- Specify generator polynomial = rsgenpoly(N,K)
- non-Punctured code

A value for the primitive polynomial was not chosen and the default block value is defined. This enables the model to be reused for several codes without the need of change the parameters. However, it must be verified if the default value is correct. Lets assume a RS(255,K). The default polynomial is obtained from the MatLab command *primpoly(ceil(log2(255+1)))*, which results in the polynomial $D^8 + D^4 + D^3 + D^2 + 1$ or in decimal notation, 285_{10}. According to [46], $100011101_2 = 285_{10}$ is a binary primitive polynomial of degree 8, which is the same as the default of MatLab.

The MatLab function *rsgenpoly(N,K)* returns the narrow-sense generator polynomial of a RS(N,K) which coefficients are roots of the default primitive polynomial. Punctured codes will not be used in the simulations. Note that, at the output of the decoder, the frame length takes the original value.

In the Figure 4.6 are shown the results for RS codes, in particular the RS(255,K), for a BSC. The values of K that were simulated are [165, 181, 197, 213, 229, 245].

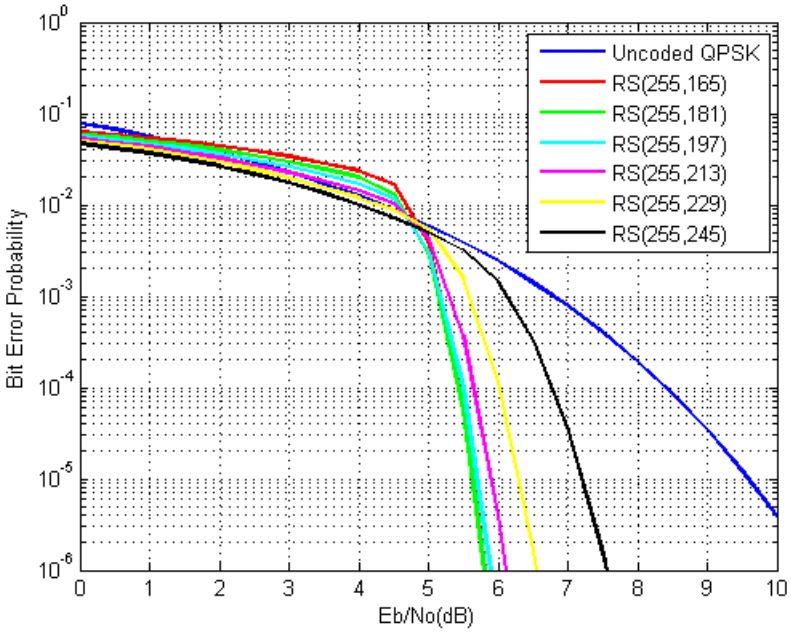

Figure 4.6: RS(255,K) with random errors Simulink model simluation results

As the code correction capability $(N - K)/2$ increases, the BER at the output decreases. For each code, there are a certain lower E_b/N_0 region that it is not viable to use it.

It is crucial to validate this results by comparing them with the expected BER. In Figure 4.7 is shown the result for the RS(255,213) along with the estimated value from both MatLab *bercoding* and the analytical estimation from equations 3.17 and 3.18. This last estimation is computed in the MatLab script shown in Appendix B.

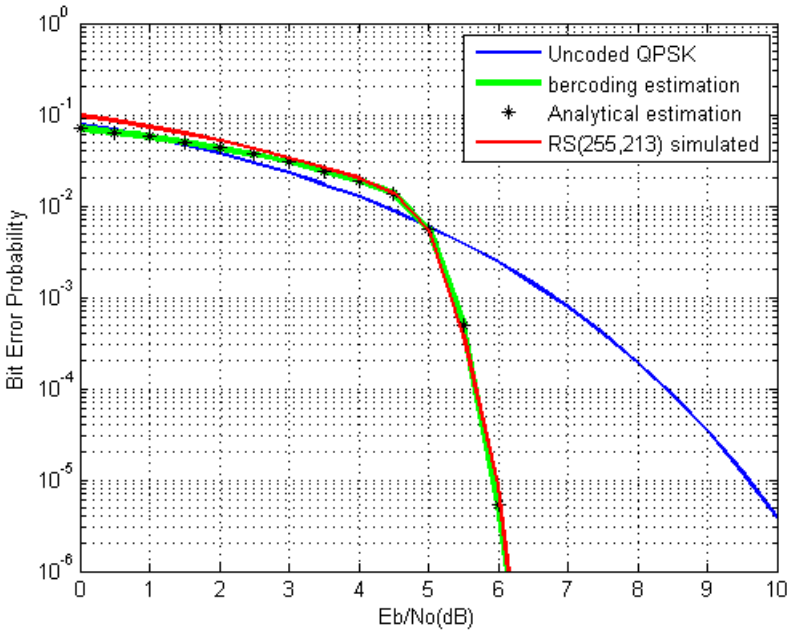

Figure 4.7: RS(255,213) with random errors Simulink model simluation results

According to the results, the Simulink Reed-Solomon model BER value is close to the expected value. Note that, for higher E_b/N_0 values, the simulated BER deviates from the theoretical value. This is due to the approximation which establishes that $P_{eb} \approx P_{es}$.

Taking into account the random RS model, as well as the burst error generation described in 4.3, a burst error model for RS codes was also created. The combination of both models results in a new model represented in Figure 4.8.

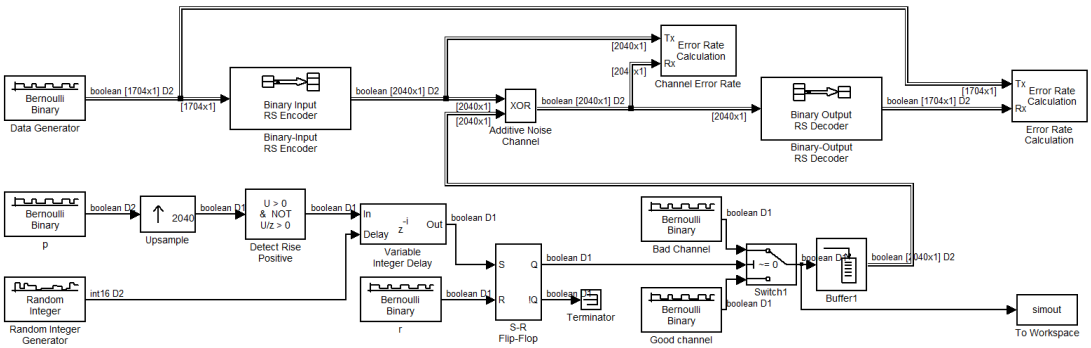

Figure 4.8: Reed-Solomon simulink model in presence of burst errors

In this model, the generation, coding and decoding of the information are equal to the previous one. However, to ensure that each codeword is only affected by one burst error, for simulation purposes, some blocks were added into the random errors generation. First of all, a channel transition to a bad state (burst errors) can only occur one time per codeword

(the sample time of this block is $t_s \times R \times m \times N$). The *Upsample* and *Detect Rise Positive* guarantee a sample rate equals to the bit rate of the codeword. In order to prevent the burst errors to always start at the beginning of the codeword, a *Variable Integer Delay* block along with a *Random Integer Generator* controlling the delay was introduced. In this way, the step created by the *Detect Rise Positive* is delayed to a random bit in the codeword, between zero and $(N \times m) - Burst_{length}$. At any time, the burst error can be stopped with a probability r. The following blocks shall be familiar to the reader, according to the model of burst errors generation. The *Buffer* block at the output of the error generator groups $N \times m$ bits to match the length of the codeword. An additional block *To Workspace* stores all the error bits for later statistic analysis.

In the Figure 4.9 are the results for RS codes in presence of burst errors. This simulation considers an average burst size of $m \times (N - K)/2 = 168$ bits.

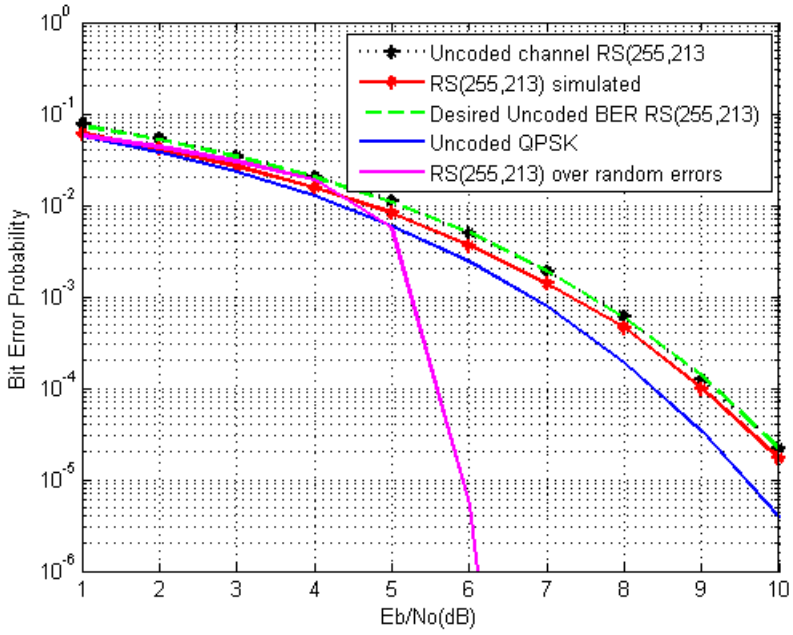

Figure 4.9: RS(255,213) with burst errors Simulink model simluation results

The statistical analysis of the burst errors for 7dB of E_b/N_0 is listed below:

- Number of burst errors: 1295
- Average burst length: 172 bits
- Maximum burst length: 1088 bits
- Minimum burst length: 2 bits

As expected, the RS can not correct all errors since there are burst errors in the channel which the length is above the code correction capability. Furthermore, the coding gain is the

same independently of the channel error probability. An improvement to previous model is represented in Figure 4.10.

This model introduces Block Interleavers to make the system more robust in presence of burst errors, as explained in Section 3.7. The choice of using a Block Interleaver instead of a Convolutional Interleaver was due to the correction characteristics of RS codes. Thus, in a scenario that a burst error occurs in the channel, after the Deinterleaving process there will be no real random error pattern. Instead, the burst error will appear distributed by all the interleaved codewords, in smaller burst errors. Since RS corrects t symbols (groups of m bits) independently of their location (randomly or sequentially distributed), burst errors affecting $t \times m$ bits will not be a problem in the RS codeword correction process. Furthermore, if a pure random error pattern was obtained after the Deinterleaver, RS codes could perform worse than in the small burst error scenario.

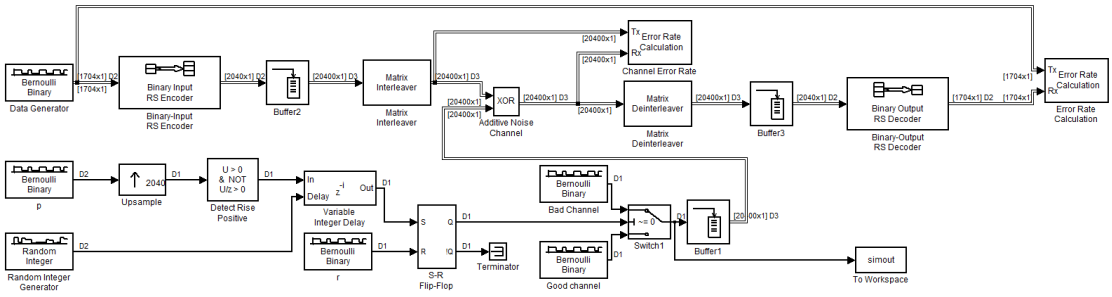

Figure 4.10: Reed-Solomon with Block Interleaving simulink model in presence of burst errors

For instance, lets consider a RS(255,213) code, which can correct up to 21 symbols, correspondent to 168 bits. If a random error of 100 bits occurs, in the worst scenario it could corrupt up to 100 different symbols (one bit for each symbol), exceeding the code correction capability. However, if it is a burst error, in the worst case only 14 symbols affected by errors which is perfectly correctable by a RS(255,213) code. Therefore, a Block Interleaver performs better together with RS codes when compared to a Convolutional Interleaver, which could lead to a more random error pattern and degrade system performance.

Since the block *Matrix Interleaver* requires that the input signal have the total data to fill the matrix, a *Buffer* was inserted after the *RS Encoder*. In this particular scheme, the Interleaver has 10 rows, which can store 10 RS(255,K) codewords. The buffer size at the error generator output was also updated to match the number of bits inserted in the channel. At the receiver side, after the *Matrix Deinterleaver* block, a buffer was used to change the frame size in order to match the required length for the *RS Decoder*. One last modification was made in the *Receive Delay* parameter in the *Error Rate Calculation* block. The used interleaver and deinterleaver blocks does not introduce latency, as well as the buffer in the receiver (since it just groups $N * m$ bits from a larger frame). Thus, the delay is given only by the buffer in the emitter. Therefore, the total latency is the number of data bits per codeword multiplied by the number of codewords stored in the buffer. The results of this model can be seen in Figure 4.11.

This results considers a block interleaver with 10 rows, that means it mixes the information of 10 different codewords. The graph shows a significant performance improvement when compared to the burst errors model without an interleaver. Above 5.5dB the model did not

found errors. This is due to the simulation precision.

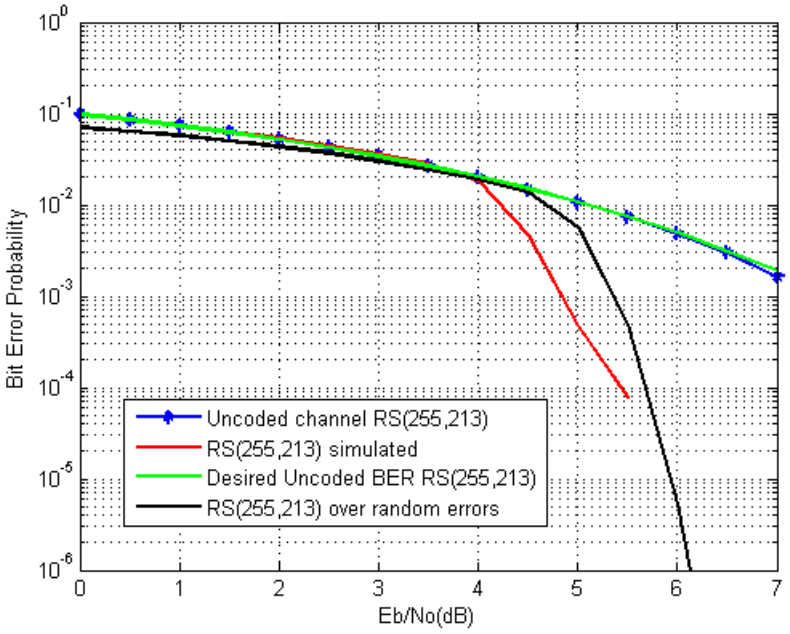

Figure 4.11: RS(255,213) + Block Interleaving with burst errors Simulink model simluation results

The study of Convolutional codes implies also an implementation of Simulink models. The Convolutional system model for random errors is shown in Figure 4.12.

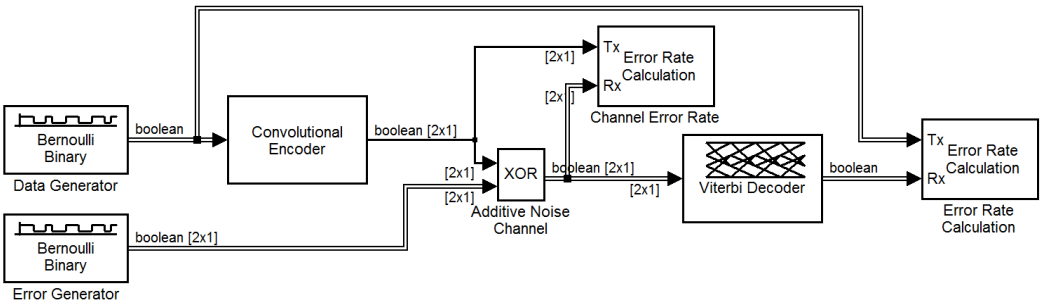

Figure 4.12: Convolutional codes simulink model for random errors

Once again, the random error model in was used and a *Convolutional Encoder* and a *Viterbi Decoder* were introduced in the data path to provide error correction. The figure describes a system using a Convolutional code with constraint length $K = 7$ and a code rate $R = 1/2$ which means that for each input bit, two parity bits are generated at output. The code uses two generator polynomials:

$$g1 = 171_8 = (1111001)_2 \tag{4.2}$$

$$g1 = 133_8 = (1011011)_2 \tag{4.3}$$

The equivalent encoder schematic for this code is shown in Figure 4.13, where each block Z^{-1} is a shift register and the the sum blocks are in fact modulo 2 adders. The input signal is *x(t)* and the outputs of the modulo 2 adders are *y1(t)* and *y2(t)*.

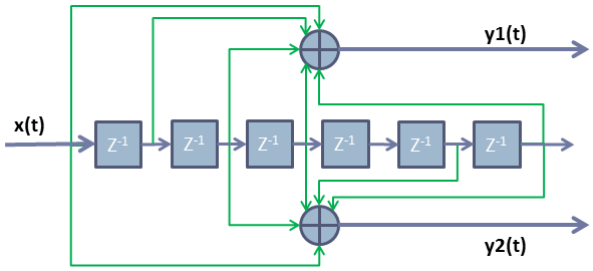

Figure 4.13: Convolution Encoder for K=7, R=1/2 and Generators 171_8 and 133_8

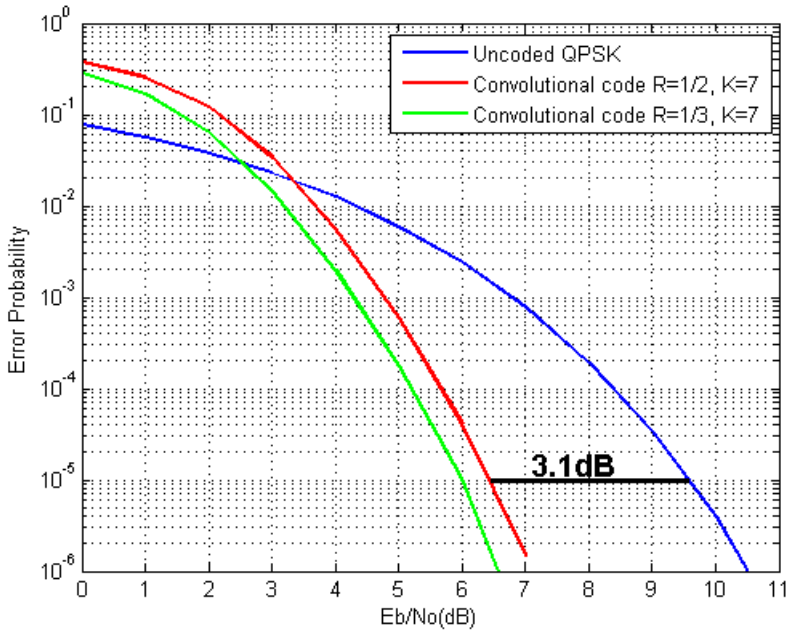

Figure 4.14: Convolutional codes with random errors Simulink model simluation results

The configuration of *Convolutional Encoder* requires a "Trellis Structured" which is obtained by the MatLab function *poly2trellis(7, [171 133])*. The operation mode was configured to *Continuous* mode which means that the encoder do not force its internal state to the initial state. At the *Viterbi Decoder* the code was configured as in the encoder as well as the operation mode. Since the input symbols have only two levels (0 or 1), the decision type was set to *Hard Decision*. The *Traceback Depth* was set to 41 according to Forney's results [46],

which shows that values above $5.8K$ presents negligible error probability due to codeword truncation in the decoder. The *Error Rate Calculation* receive delay is configured to be the same as the *Traceback Depth*, since this parameter decides how many branches the decoder has to store before it starts to decode the received data. The *Error Generator* was configured to generate bits R times faster than the *Data Generator* and the output is a frame with size $1/R$.

In Figure 4.14 is shown the results for Convolutional codes with $K = 7$ for both $R = 1/2$ and $R = 1/3$ in the presence of random errors. At 10^{-5} of error probability, the coding gain for the Convolutional code with $R = 1/2$ is 3.1dB, which is the expected value according to the Table 3.3 (5.1dB-2dB). The performance of convolutional codes for lower values of SNR is worse than the performance of RS codes for the same scenario.

A burst error model for Convolutional codes is obtained by replacing the Error Generator in the previous model by a burst error generator, resulting in the model shown in Figure 4.15.

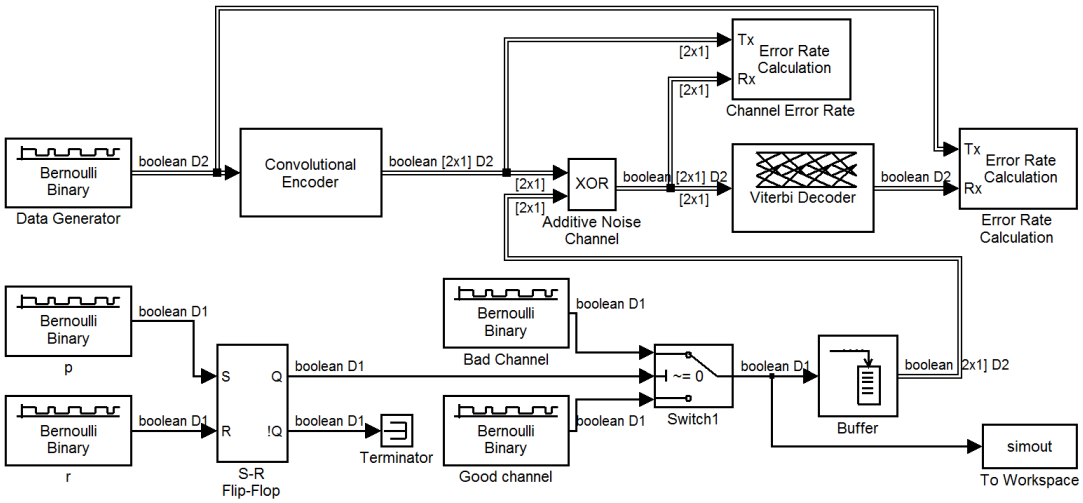

Figure 4.15: Convolutional codes simulink model in presence of burst errors

The results for Convolutional codes for burst errors are represented in the Figure 4.16. It can be seen that the performance in presence of burst errors is severely degraded.

As in the RS scenario, the usage of Interleaving techniques shall offer an improvement in the system Bit Error Rate. Therefore, it shall be created a model including an Convolutional Interleaver together with Convolutional codes. Since the Convolutional codes, as oposed to RS codes, does not perform well in presence of burst errors, the most suitable interleaving technique is the Convolutional Interleaver, which can spread the bits more efficiently than the Block Interleaver.

The number of shift registers used was $B = 14$ in order to spread the bits in a range larger than 4 Traceback lengths. The delay introduced by the interleaving is, as referred in Section 3.7, B(B-1). At the output of the Viterbi decoder, this latency is $B(B-1)R$, since the duration of each bit is R times than its duration in the channel. The implemented model is shown in Figure 4.17.

As the reader can observe in Figure 4.18, the BER curve for Convolutional codes with a Convolutional Interleaving approaches the BER curve for this codes in presence of random

errors. The parameters of the used Convolutional Interleaver are:

- Number of Rows of shift registers: Traceback_length/2
- Register length step: 1/R

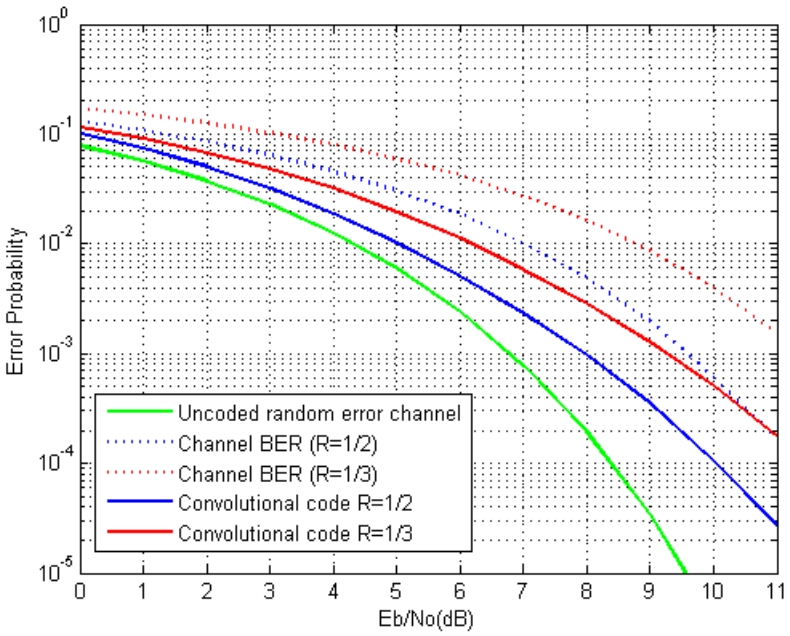

Figure 4.16: Convolutional codes with random errors Simulink model simluation results

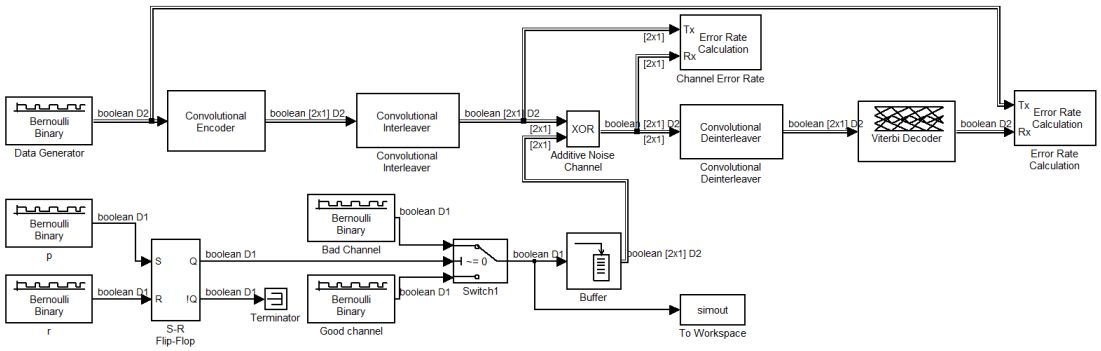

Figure 4.17: Convolutional codes with an Convolutional Interleaver simulink model in presence of burst errors

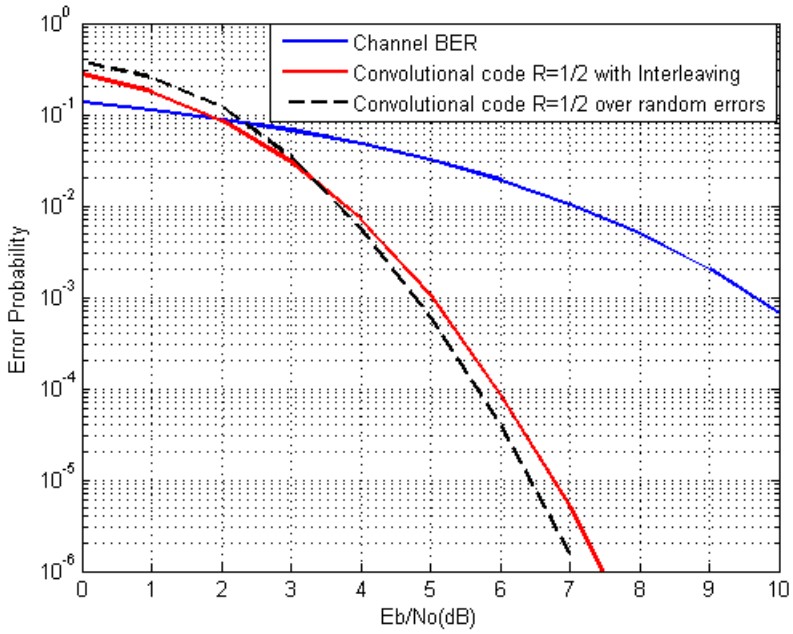

Figure 4.18: Convolutional codes + Convolutional Interleaving with burst errors Simulink model simluation results

4.2 Synchronous Implementation

In this section will be described the implementation of RS codes, Convolutional codes, Interleaving and CRC-32 models in a synchronous system. Later, these models will be converted into an asynchronous architecture. The System Generator synchronous models were derived from the previous studied Simulink models, replacing the encoder and decoder section by blocks from the System Generator Blockset library.

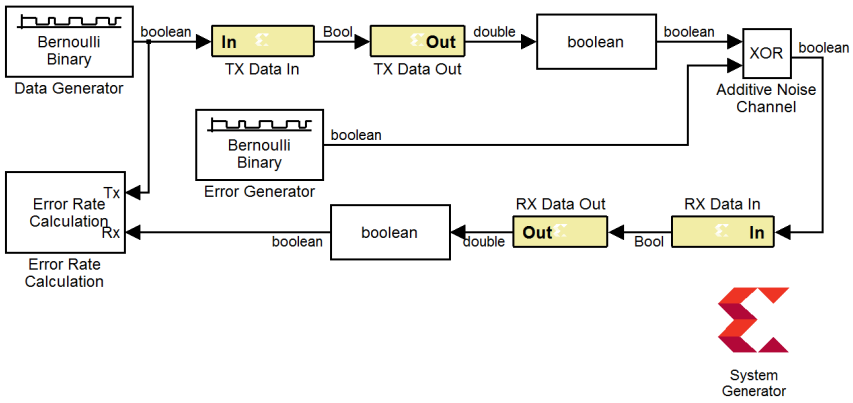

Figure 4.19: System Generator random error channel model

As an introduction of System Generator design method, a basic BSC model, in Figure 4.19, will be presented.

Before continuing to the model description, the System Generator token should be analysed. It makes the link between Simulink and the Xilinx software and its configurations are shown in Figure 4.20.

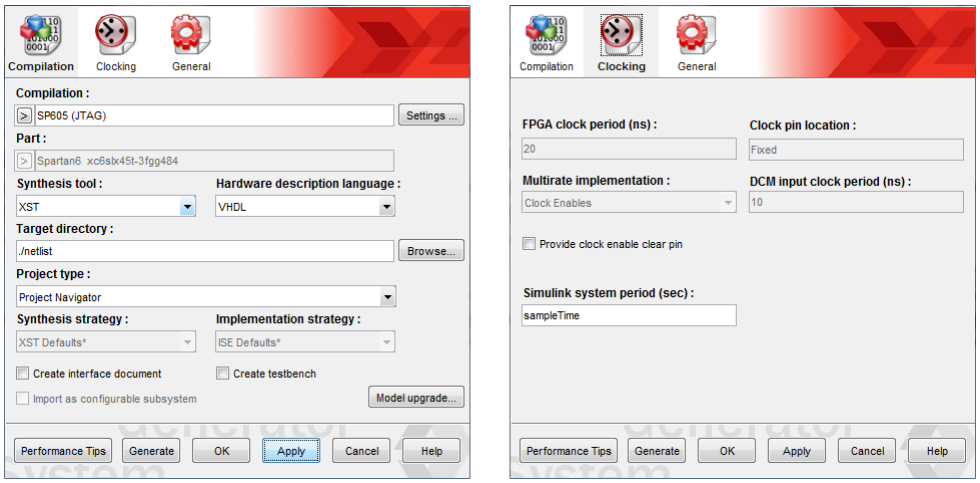

Figure 4.20: System Generator Token block

In the left figure are shown the configurations regarding the model compilation. The compilation parameter was set to "Hardware co-Simulation" for the SP605 board, the Synthesis Tool to use is the "XST" and the Hardware Description Language was chosen to be VHDL. In the Clocking parameters (on the right) it shall be setted a Simulink System Period. This value makes the conversion between the FPGA sample time (20ns) and the Simulink sample time. The System Generation token block invokes all necessary procedures to synthesise the HDL code from the designed model.

Comparing this model to the the random error channel model in Figure 4.1, the reader can observe that I/O blocks from the Xilinx Blockset library were added to the existing *Data Generator*, *Error Generator*, *XOR* and *Error Rate Calculation* blocks. The blocks *TX Data In* and *RX Data In* are the data inputs to the emitter and receiver, respectively. In both, the parameter *Output Type* was set as Boolean and the *Sample Period* was defined to have the same value of the input data sample time. On the other hand, the blocks *TX Data Out* and *RX Data Out* are the FPGA data outputs. The additional *Data Type Conversion* blocks converts the data type from the gateway outputs, which are in double format, to boolean in order to maintain the data type model coherence.

In Figure 4.21 is shown the channel error probability for the model described in Figure 4.19.

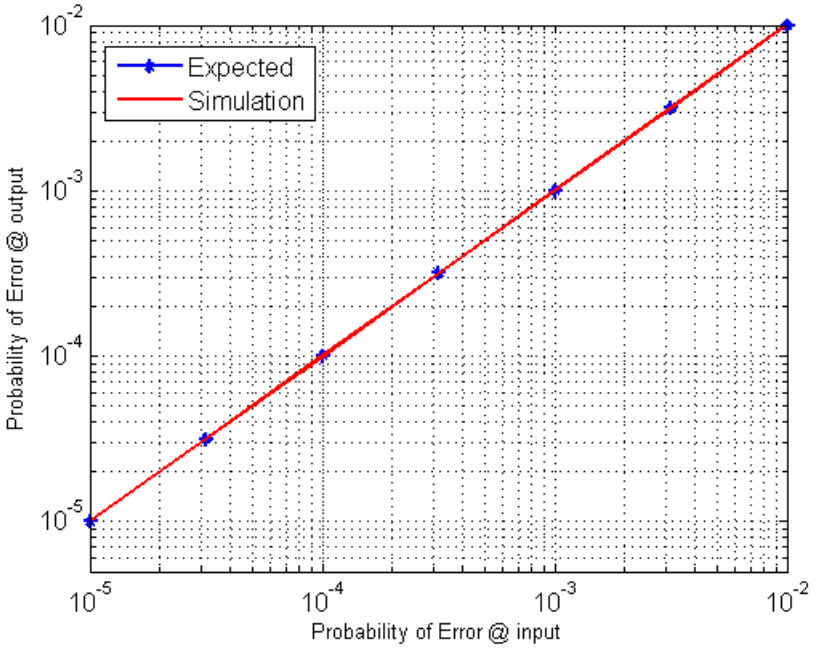

Figure 4.21: BSC System Generator simulation results

As expected, the error probability of the channel approaches the desired value.

4.2.1 Reed-Solomon codes

Taking in consideration the Simulink model of Reed-Solomon codes on an BSC in Figure 4.5, the equivalent System Generator model is shown in Figure 4.22. A brief look reveals that the differences between them are not only in the encoder and decoder blocks. In order to guarantee the proper functioning of the system, additional blocks must be inserted in the model to control the main encoder and decoder blocks.

The connection colours represents different sample times: the red paths operates at the bit generation sample rate and the green ones have a sample rate eight times slower, since they are related to symbol rate.

The *Reed-Solomon Encoder 8.0* block has three inputs: *input_tdata_data_in* which is the RS symbols input; input_tvalid, which indicates if the symbol at the data in port is valid; finally, *input_tlast* that is used to mark the last symbol of the input block in order to generate events at the output. The outputs of RS encoder block are: *input_tready* indicates that the encoder is ready to receive new symbols; *output_tvalid* signal, which is set if the block has valid data at the output; *output_tdata_data_out* is the data output; *output_tlast* signals the last symbol of the last codeword; *event_s_input_tlast_missing* and *event_s_input_tlast_unexpected* are event outputs and indicates if *input_tlast* is not according with the expected value.

Since Reed-Solomon encoder block is a generic RS encoder, its parameters must be configured. The implemented code is a RS(255,213), which is not a standard. Thus, the code parameters are:

- Code Specification = Custom

- Variable Number Of Check Symbols (r) = unchecked
- Variable Block Length = unchecked
- Symbol Width = 8
- Field Polynomial = 285_10
- Scaling Factor (h) = 1
- Generator Start = 0
- Symbols per Block (n) = 255
- Data Symbols (k) = 213

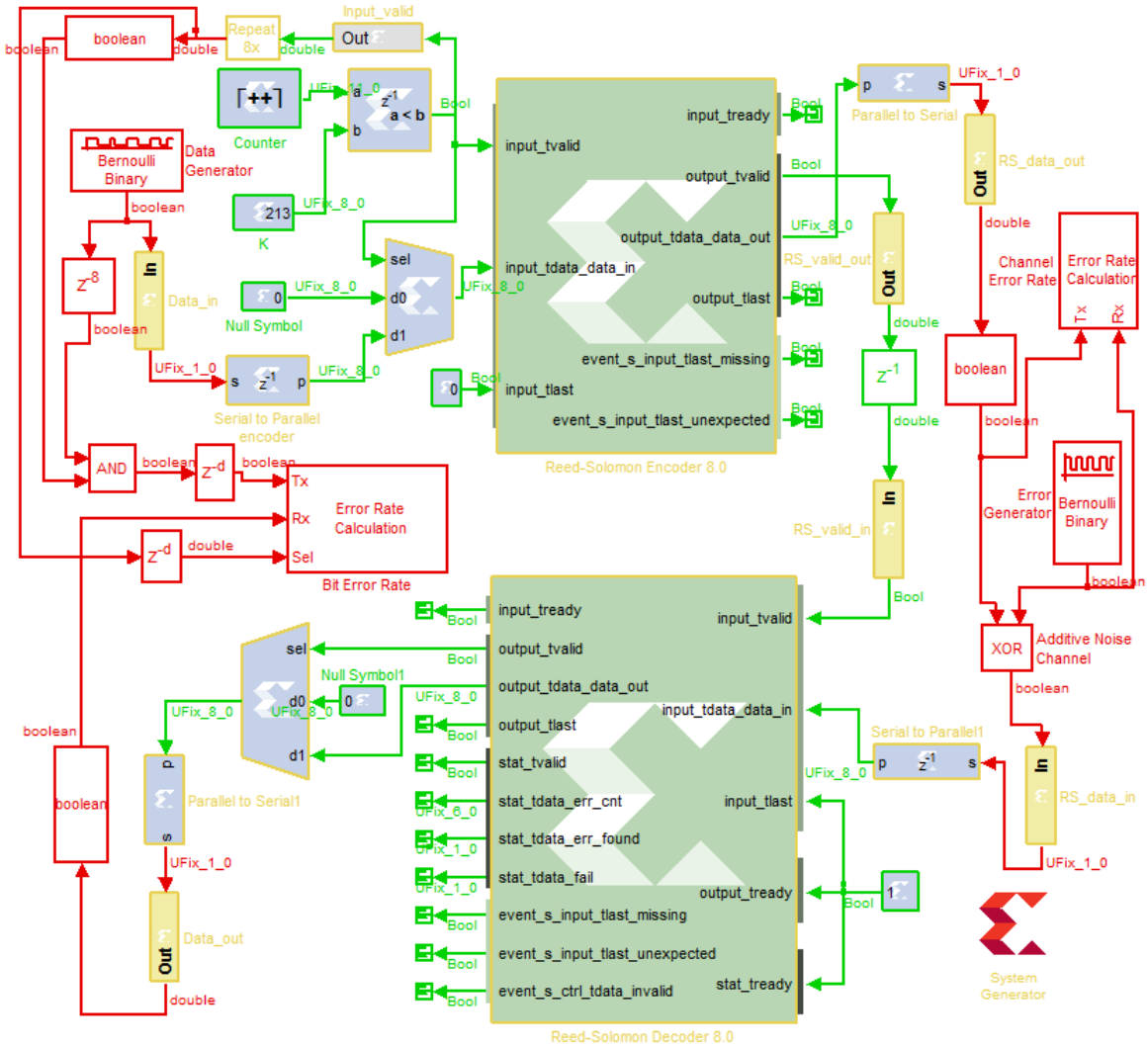

Figure 4.22: System Generator Reed-Solomon model

The *Data Generator* block produces bits at a determined rate. The *Data_in* translates the received data into an unsigned fixed-point one bit signal. The following block, *Serial to Parallel*, takes eight serial bits and groups them in parallel bits at the output, being the first input bit the least significant one. These parallel bits will be the input data symbols. It is important to note that this block introduces a latency equals to the period of eight bits. Next, it shall be generated the *input_tvalid* signal. This is made by three blocks: a counter that counts cyclically from zero to a predefined number (that will be further discussed) with a period of a symbol (eight times the bit period in the Data Generator); a constant block storing the number of the desired data symbols (K); and a relational block that sets its output if the value of the counter has a lower value than K. To synchronize this last block output to the symbol input in the encoder, one delay (eight input bits duration) was inserted in it. The signal generated for *input_tvalid* also controls a multiplexer that switches from input data to a null symbol (eight zero bits) when all required data symbols were received by the encoder. Since in this implementation the events are not an important aspect, *input_tlast* is set to low.

The output of the encoder passes through a *Parallel to Serial* converter in order to perform a serial bit transmission (first output bit is the least significant one). After this conversion a gateway out (*RS_data_out*) sends the information to the Additive Noise Channel. Once again the data type conversion to boolean is made after the gateway out. The model of the channel is identical to the previous studied one, having an *XOR* to change the transmitted bits every time that the *Error Generator* produces a "one" bit. Along with the data transmission, a synchronization bit is also transmitted in order to signal the receiver if the data at its input is valid. This signal is delayed by one symbol period in order to compensate the latency introduced by the Serial to Parallel block at the RS decoder input.

In the receiver side, the input data is converted from bits to symbols (eight bits). The *Reed-Solomon Decoder 8.0* has five input signals: *input_tvalid* to provide information about the validity of the input data; *input_tdata_data_in* is the RS symbols input; *input_tlast* signals if the received symbol is the last one, used to generate events at output; *output_tready* signals the decoder if the following block is available to receive data; *stat_tready* receives information about the availability of the statistical processing unit in order to send statistical information related to the decoding process to a dedicated module.

This block has several outputs divided in three categories: data output, statistical output and event signalling.

The *output_tvalid* provides information about the validity of output data; *output_tdata_data_out* is the decoded RS data symbols; *output_tlast* signals goes high when the last symbol of the last codeword is present in the output data port. The *input_tready* informs if the block is available to receive new data.

The *stat_tvalid* is set when the information at the statistical output is valid; *stat_tdata_err_cnt* provides the number of error that were corrected in the decoded codeword; *stat_tdata_err_found* is setted if any errors or erasures are detected in the current received codeword; *stat_tdata_fail* is setted if, in the current codeword, the decoder was unable to recover all the information symbols. One last remark in this last parameter is that for RS codes is possible in many cases to detect if the error correction capability (t) was exceeded. However, some cases are not detectable and it is required the usage of other techniques to validate the decoded data.

The event signals have a similar behaviour as in the encoder block. An additional event port is available, called *event_s_ctrl_tdata_invalid*, however it is only used when a variable codeword length, number of check symbols or a punctured code is implemented.

The remaining input control ports are set to high to enable the block output. Note that *input_tlast* can be set low or high if the event outputs will not be considered. At the output of the decoder, a multiplexer controlled by the *output_tvalid* port determine if the output is the information symbols or null symbols. After that, the symbols are converted to serial bits that constitutes the output of the receiver. The Error Rate Calculation takes the received bits and the input bits masked by the *Input_valid* signal in the emitter to measure the system BER. In order to consider only the transmitted data, an additional port was enabled in this block, *Sel*, that is operated by the *Input_valid* signal. An important remark in this signal is that its sample rate is converted to match the bit generation rate. Furthermore, the RS decoder block must be configured with the same code parameters established in the encoder.

With regards to the counter maximum value, as a first approach, the 254 seems a reasonable value, since the codeword has 255 symbols and the counter starts from zero. However, it may not work for all codes due to System Generator RS blocks constraints. These blocks are based in the LogiCORE IP Reed-Solomon Encoder and Decoder, specifically the version 8.0. The Product Guide of the LogiCORE IP Reed-Solomon Decoder 8.0 establish that the processing delay (PD), in number of clocks, for a particular RS(N,K) code is given by:

$$PD = 2t^2 + 9t + 3, \quad t = \frac{N-K}{2} \tag{4.4}$$

In order to the decoder provide a continuous decoding operation, PD shall be lower or equal to the number of codeword symbols. If this condition does not occur the decoder can not guarantee the correct behaviour. Despite this LogiCORE possesses input buffers, if the input codewords presents an higher rate than the core processing capabilities, eventually the buffers fills, discarding newer codewords. The maximum throughput achieved in a continuous operation is the clock frequency (MHz) times the number of bits per symbol, in Mb/s. If $PD > N$ then the maximum throughput is approximately $(N/PD) \times clock\_frequency \times symbol\_length$ Mb/s [50]. It means that when the decoder is configured to a RS(255,213) code, $PD = 1074$ and the maximum bit rate is $\approx$ 95Mb/s, considering a 50MHz clock frequency. In order to avoid data loss, the counter in the emitter was configured with a count limit of 1073. Such value, forces the RS Encoder to wait 1074 symbol periods between consecutive codewords.

Along with processing delay, latency is also present both in the Encoder and Decoder. In contrast with processing delay, latency only shifts the output in time, in other words, the duration of a codeword is not changed. Xilinx specifies a latency of $2 + (number of channels)$ for the RS Encoder 8.0. In this model, the encoder is only implemented with one channel, presenting a latency of 3 symbol periods. For the decoder this value is not so linear and shall be obtained in the Graphical User Interface of the Xilinx Core Generator, which is a tool to implement LogiCORE IPs with the desired characteristics. The value obtained was 1341 symbol periods. Additionally, the *Serial to Parallel* blocks inserted in the encoder and in the decoder presents a delay of one symbol each. In order to synchronize the input data and output data as well as the signal *Input_valid* at the *Error Rate Calculation* input, two delay blocks were inserted. The delay value of each one is $8 * (1341 + 3 + 1)$ (in bit periods) which is, in symbol periods, 1341 from the decoder, 3 from the encoder and one from the *Serial to Parallel Decoder* block. Note that a previous delay block immediately after *Data Generator* synchronizes the data bits with Input valid signal, with a value of one symbol period.

The performance of System Generator model for a RS(255,213) code over a BSC can be observed in Figure 4.23. The results shows that the model performs as expected.

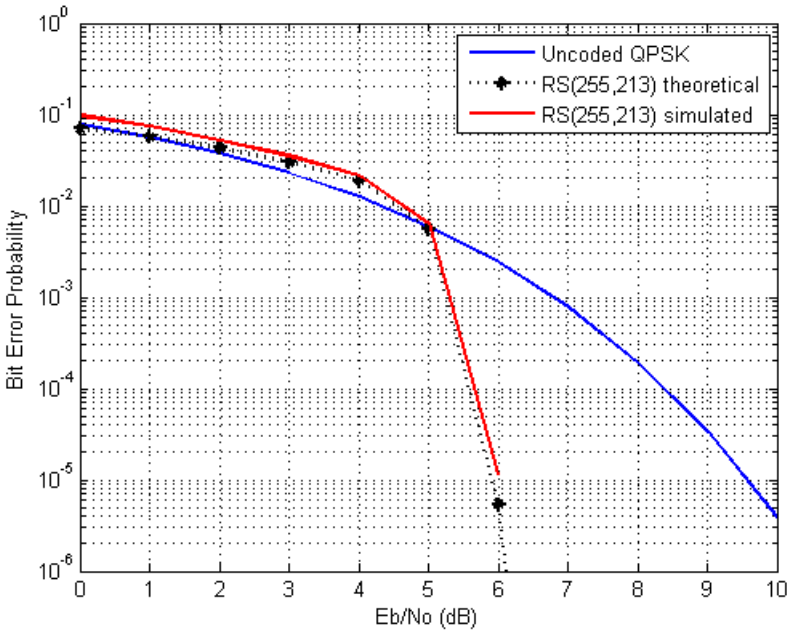

Figure 4.23: RS(255,213) System Generator model simluation results

4.2.2 Convolutional codes

The implementation of Convolutional codes in System Generator is shown in Figure 4.24. The model considers a Convolutional code with $R = 1/2$ and $K = 7$. The colours shown in the model represents the sample times for each signal.

As in the previous models, the block *Data Generator* generates bits at a determined sample time. These bits are read into the FPGA in the *Data_in* gateway that feeds the *Convolutional Encoder 8.0* in the port *data_tdata_data_in*. This encoder requires also a signal to validate the information in the data input port that was set continuously to high (non-sampled constant). The three outputs of the encoder are: *data_tdata_data_in* which is the output of encoded data, *data_tvalid* which indicates that the output data is valid and *data_tready* which signals that the encoder is ready to accept new data.

The parameters of this block are listed below:

- Punctured = unchecked
- Output rate = 2
- Constraint length = 7
- Convolution code0 = '1111001'
- Convolution code1 = '1011011'

Since the encoder output is formed by two bits in parallel and the channel is a BSC, a conversion *Parallel to Serial* is made before the encoded signal is sent to the gateway out,

Conv_out. Note that after this conversion the sample rate is two times faster than in the original data. The encoded bits are then converted to boolean and sent through the BSC model.

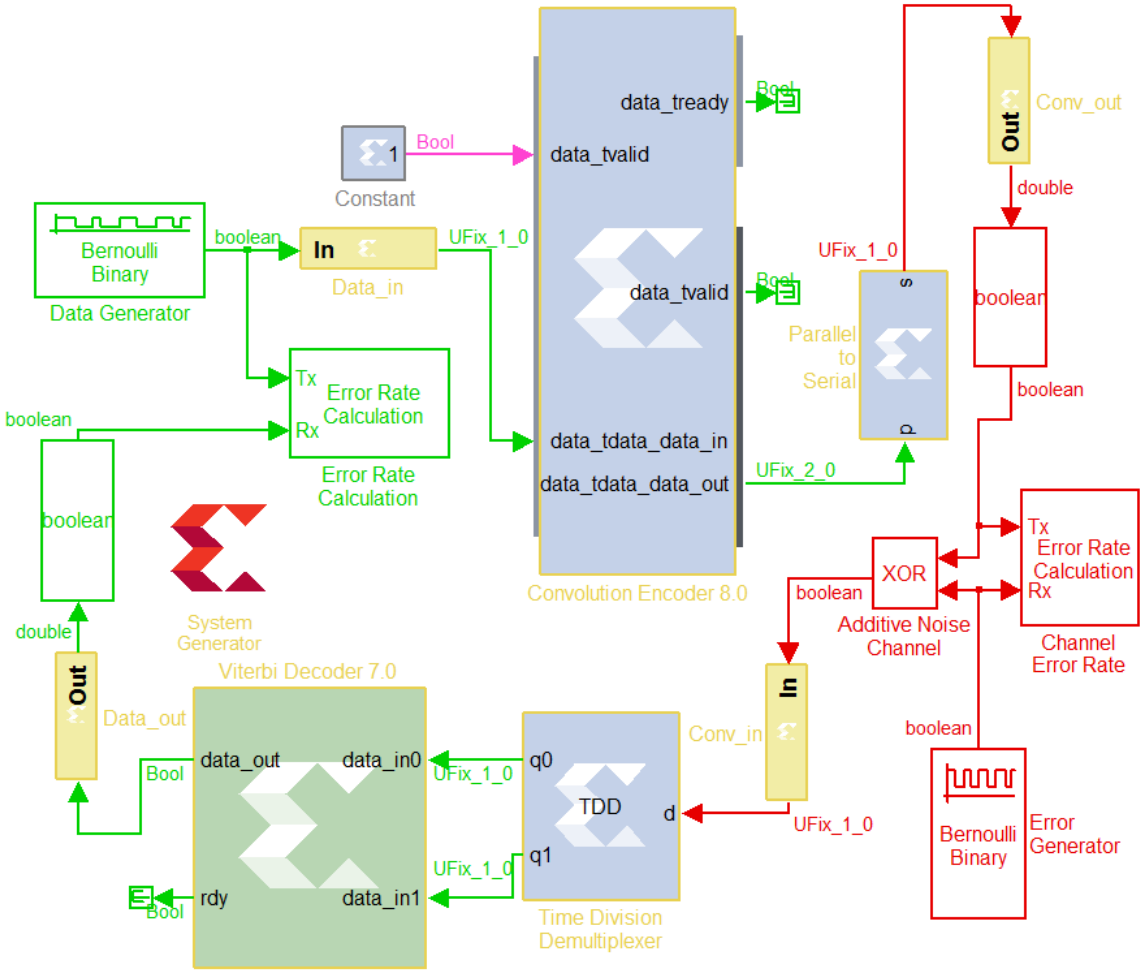

Figure 4.24: System Generator Convolutional synchronous model

In the receiver side the encoded data is read into the FPGA in the gateway *Conv_in*. Since the Viterbi decoder used has disjoint input ports for the n bits generated by the n generators in the encoder, a *Time Division Demultiplexer* block was inserted in the receiver. The implementation type of this block is multiple channel and the "Frame sampling pattern" configured was [11], which means the block has two outputs and consecutive bits shall be routed to different output ports. This block presents a latency of 2 input bit periods that corresponds to one bit period of the original data.

The decoding process is preformed by the *Viterbi Decoder 7.0* block. As previously said, the input ports *data_in0* and *data_in1* are the two encoded bits for each generator. The output port of the decoder is *data_out* and *rdy* port signals if the output data is valid. The Viterbi Decoder parameters are:

- Viterbi Type = Standard

- Constraint length = 7
- Traceback length = 42
- Architecture = Parallel
- Coding = Hard coding
- Output Rate 0 = 2
- Convolution 0 Code 0 = '1111001'
- Convolution 0 Code 1 = '1011011'

The Parallel architecture was selected due to the usage of an Hard Decoder, which is not implemented in the Serial architecture. Despite Parallel architecture requires a larger space in the FPGA, this decoder is faster than the Serial decoder. The decoded data is then sent through the *Data_out* gateway and posteriorly converted to boolean to be compared with original data in the *Error Rate Calculation* block. According to LogiCORE IP Convolutional Encoder v8.0 Product Guide, the latency of the encoder is 3 sample periods [51]. On the other hand, Viterbi decoder parallel architecture latency can be calculated as $latency \approxeq 4 \times traceback\_length + K + n = 177$ [52]. Thus, the latency between the transmitted and received data is approximately $3 + 177 + 1 = 181$. The measured latency was 197.

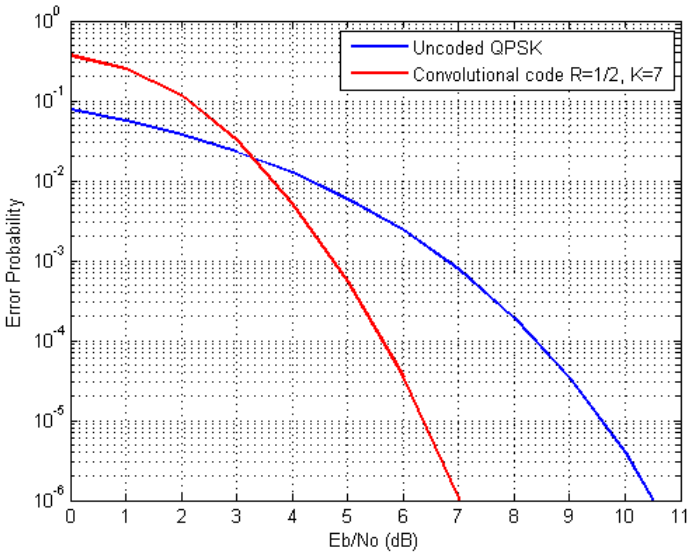

Figure 4.25: Convolutional code R=1/2, K=7 System Generator model simluation results

In the Figure 4.25 is shown the BER results for the System Generator Synchronous model for a Convolutional Code with K=7 and R=1/2. At the error probability of 10^{-5} the coding gain is approximately 3.1dB which match to the expected value.

4.2.3 Interleaving

In Figure 4.26 is represented the System Generator model for the Convolutional Interleaving. As the reader can observe, this model is identical to the model in Figure 4.19 with the addition of the Interleaver and Deinterleaver blocks.

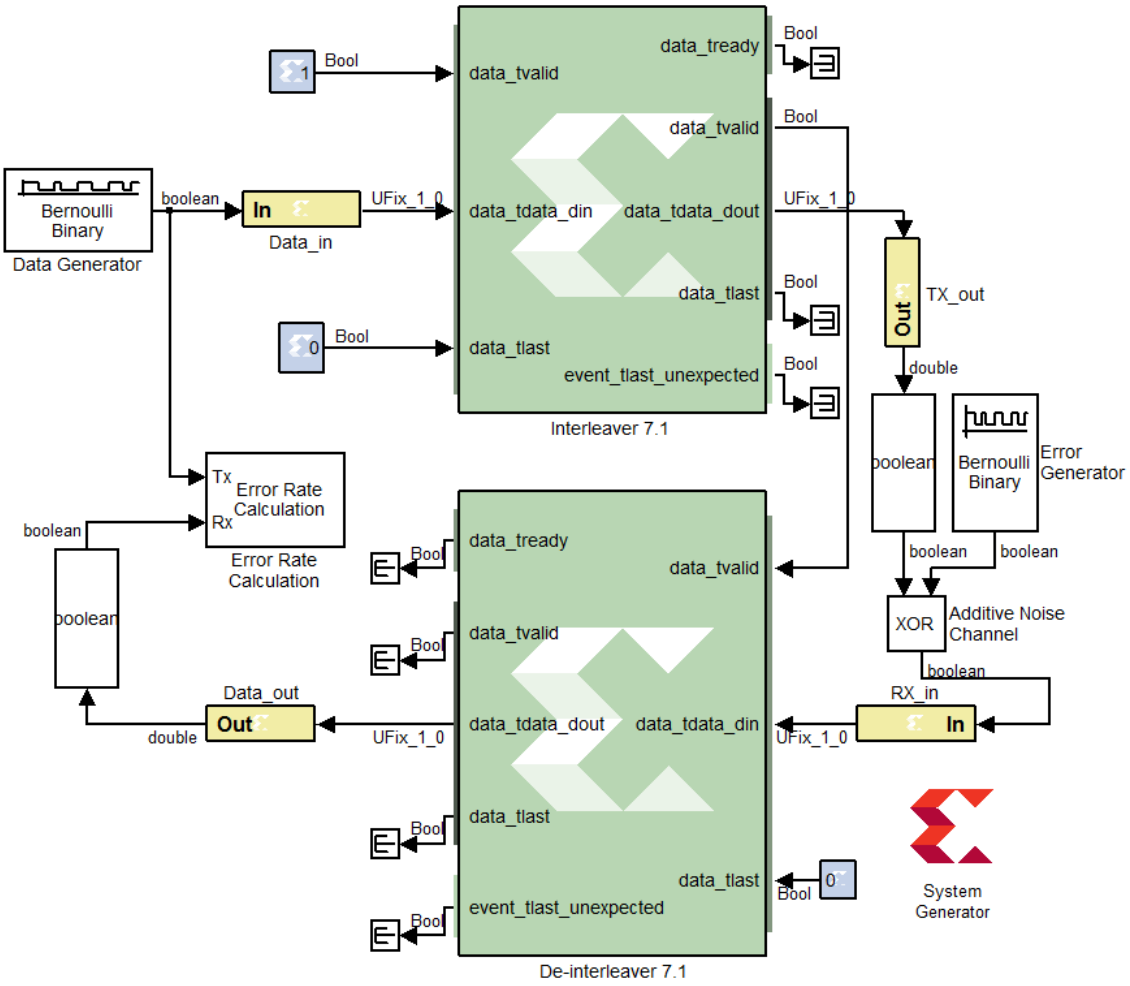

Figure 4.26: System Generator Convolutional Interleaving synchronous model

The same functional block, *Interleaver/De-Interleaver 7.1*, was used in both emitter and receiver since it provides both functionalities. This block has one input for the data to be interleaved/deinterleaved (*data_tdata_din*), one input for signalling if the data in previous port is valid (*data_tvalid*) and one other input to inform the block if the input data symbol is the last one. The output are: *data_tready*, which indicates if the block is ready to receive new data; *data_tvalid* informs that valid data is present at the output; *data_tdata_dout* is the interleaved/deinterleaved data output; *data_tlast* is asserted when a symbol is produced from the last branch of Convolutional Interleaver; *event_tlast_unexpected* is a flag that is set if the *data_tlast* signal does not corresponds to the last branch of the Convolutional Interleaver.

The parameters configured for *Interleaver 7.1* block are listed below:

- Memory Style = automatic
- Symbol Width = 1
- Type = Forney Convolutional
- Mode = Interleaver
- Symbol Memory = Internal
- Number of Branches = 14
- Architecture = ROM-based
- Number of Configurations = 1
- Length of Branches = 1 (Constant length between consecutive branches)
- Pipelinig = Maximum

Despite the "Mode" parameter, which shall be changed to "De-Interleaver", the *De-Interleaver 7.1* configurations are equal to the Interleaver. In both emitter and receiver, *data_tlast* is zero. In the emitter, *data_tvalid* is set to one and, for synchronization purposes, in the receiver this signal is generated by the *data_tvalid* at the emitter.

As previously said in section 3.7, the latency of a Convolutional Interleaver with B branches is given by $B(B-1)$. Thus, the latency for this model should be $14 \times 13 = 182$. Additionally, each Interleaver/De-Interleaver block has a latency of 6 when the pipelining is set to Maximum [53]. The total latency should be 194. However, the measured latency was 196.

In order to implement the Block Interleaving model, since the block *Interleaver/De-Interleaver 7.1* also supports Block Interleaving, the previous model was used with some modifications, shown in Figure 4.27. The parameters of Block Interleaver model are listed below:

- Memory Style = automatic
- Symbol Width = 8
- Type = Rectangular Block
- Mode = Interleaver
- Symbol Memory = Internal
- Number of Rows = 10 (constant)
- Number of Columns = 255 (constant)
- Row Permutation = none
- Column Permutation = none
- Block Size = Rows*Columns
- Pipelinig = Maximum

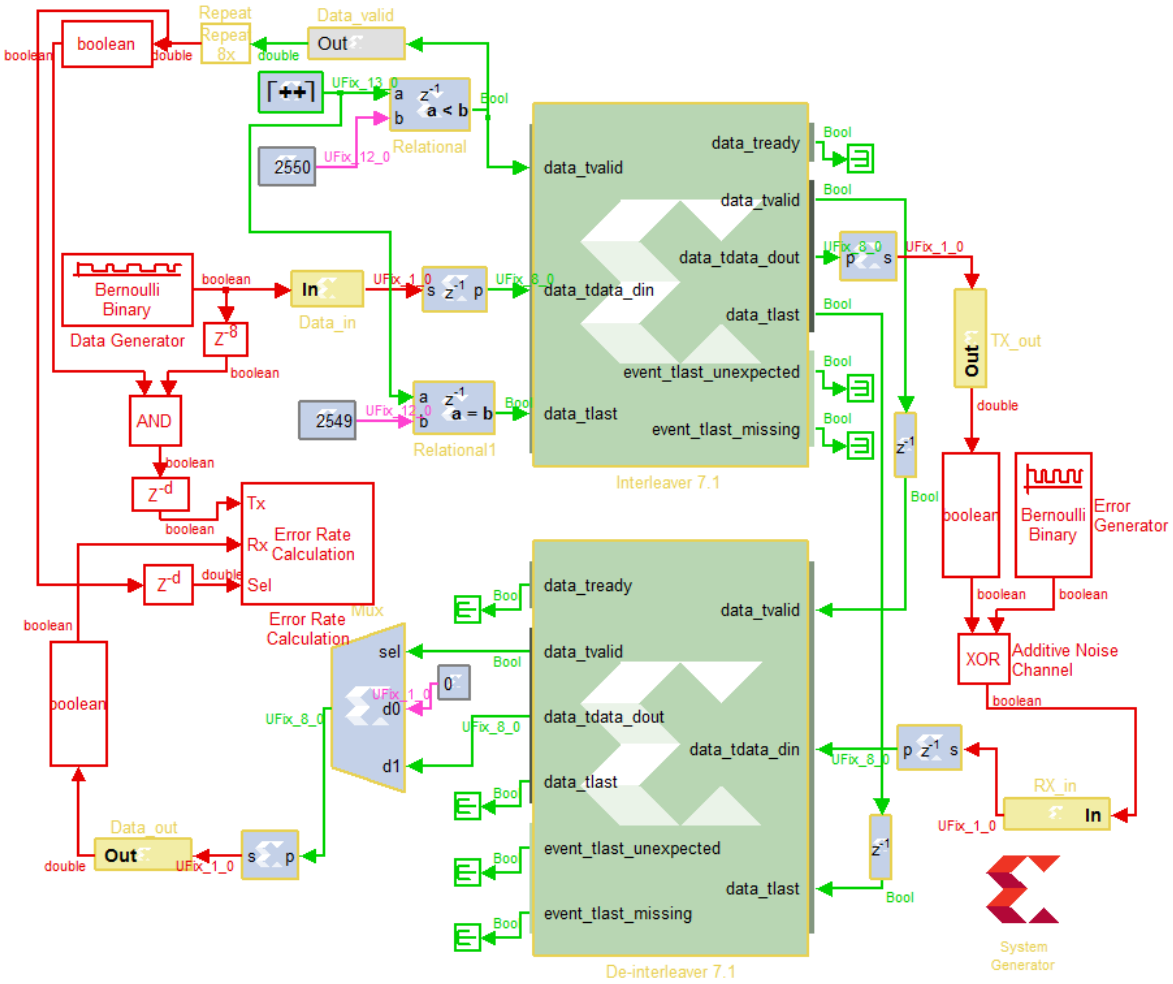

Figure 4.27: System Generator Block Interleaving synchronous model

In this implementation, the symbol width chosen was 8 in order to match the RS(255,K) symbol size. Therefore, a *Serial to Parallel* block was inserted at the input of the emitter. When operating in *Rectangular Block* mode, the *Interleaver/De-Interleaver 7.1* can only guarantee a throughput of 50% due to the usage of the same memory block to input and output data [53]. Thus, an additional control in *data_tvalid* and *data_tlast* shall be implemented. The solution is identical to the used in the RS model of the Figure 4.22: a counter, a relational block and a constant. Here, two constants and relational blocks were used in order to generate both input ports. The counter has a sample time eight times greater than the bit period and a maximum value is $255 * 10 * 2 + 10$. This means that it counts 10 RS(255,K) codewords two times, in order to give time to the Interleaver block read and write 10 codewords. The additional "+10" value is to take into account the latency of the *Interleaver/De-Interleaver 7.1* blocks. According to [53], the latency of each block is 5 symbol periods, giving a total of 10 symbol periods, which are added into the counter maximum value. Thus, the signal *data_tvalid* is generated by comparing if the counter value is lower than $10*255$ (total number of elements in the Interleaver matrix) and the *data_tlast* is a simple comparison between the counter value and $10 * 255 - 1$, since the counter starts from zero.

The output of the Interleaver is then changed from symbols to bits and goes through the BSC. The two control signals *data_tvalid* and *data_tlast* are delayed by one symbol period, due to the serial to parallel operation in the input of the deinterleaving, and feeds the inputs of the deinterleaver with the same name. At the output of the *De-Interleaver 7.1* block, a multiplexer decides, based on *data_tvalid* output port, the output of the receiver: the original deinterleaved data or null symbols. The error rate calculation is identical to the implemented in the RS System Generator model, taking the *Data_valid* signal to mask the non-transmitted bits and to enable the *Error Rate Calculation* block.

Furthermore, as in the case of RS model, the probability of error shall be multiplied by 2 in order to compensate the unusable bits in the channel.

4.2.4 CRC-32

Since the System Generator does not provide a CRC block, the System Generator CRC-32 model requires an higher detail of implementation than the previous models. The principle behind this model is to compute in the emitter the CRC-32 remainder and send it after the transmitted data. In the receiver, the remainder of the received data (with possible existing errors) is calculated and then is compared to the sent remainder value in order to evaluate the validity of received data. The implemented System Generator model can be seen in Figure 4.28.

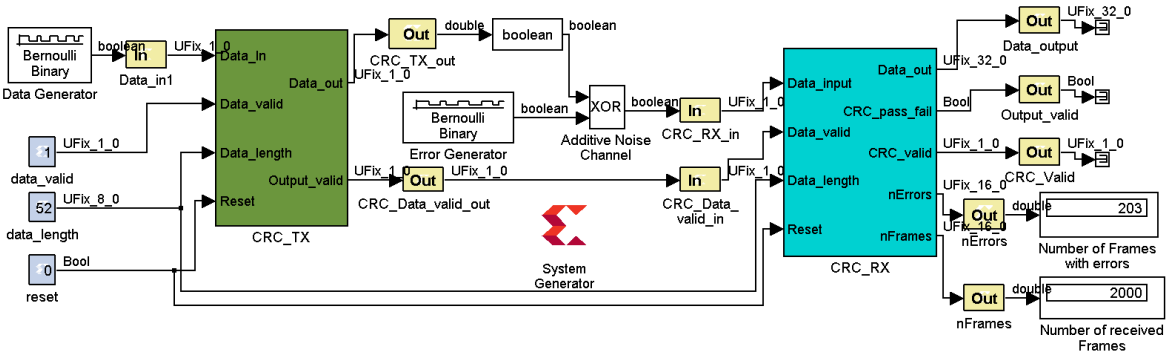

Figure 4.28: System Generator CRC-32 model

This model contains the encoder (CRC_TX) and the decoder (CRC_RX) in subsystems interconnected by the *Additive Noise Channel* model. The Figure 4.29 shows the CRC-32 encoder subsystem model.

The encoder main blocks are the *CRC_Generator* and the *crc_tx_ctrl*. In order to perform the polynomial division of modulo 2 required for CRC-32 computation, a *Black Box* was used to incorporate VHDL code of a parallel CRC-32 generator in System Generator. The simplest way to divide polynomials in modulo 2 is by using a Linear Feedback Shift Register (LFSR), as seen in Figure 4.30 [54]. This implementation requires a serial input which is not optimized since it need N clock cycles to compute the CRC remainder of a N bit input. A more efficient algorithm uses a parallel architecture which accepts parallel bits at the input and processes them on a single clock period. The parallel algorithm computes its next state based on the

previous state as well as the input data.

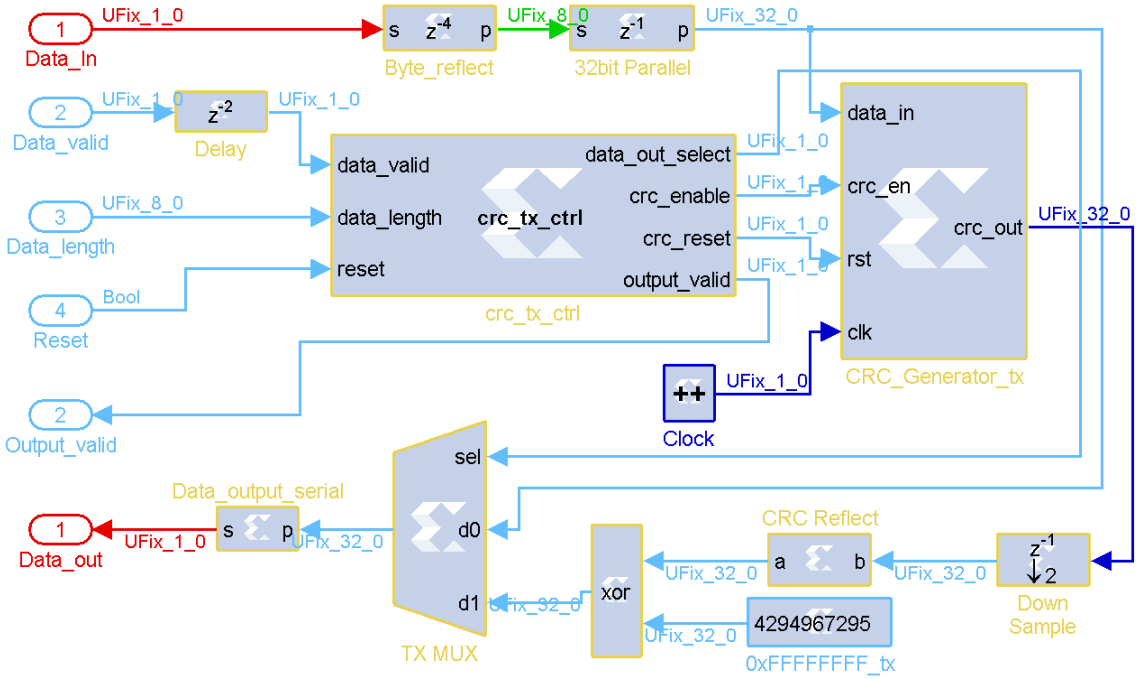

Figure 4.29: System Generator CRC-32 emitter synchronous model

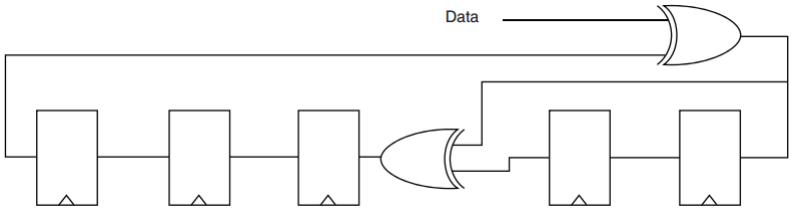

Figure 4.30: Hardware implementation of a serial CRC generator

The CRC implementation is the CRC-32 used in the IEEE 802.3 Ethernet standard. The specifications of this algorithm are listed bellow:

- Polynomial Width: 32
- Polynomial Value: 0x04C11DB7 = $1 + x^1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^{10} + x^{11} + x^{12} + x^{16} + x^{22} + x^{23} + x^{26} + x^{32}$
- Initial state: 0xFFFFFFFF
- Reflected input bits: Yes
- Reflected output bits: Yes

- Output XOR Value: 0xFFFFFFFF

A remark shall be made for the last four parameters above. This algorithm initializes its initial state to be different that zero in order to prevent blind spots on the error detecting process when bytes of input data is zeroed. The CRC-32 standard specifies that the bit order of each byte before CRC computation as well as the CRC result shall be reversed. The last parameter performs an XOR operation with 0xFFFFFFFF to obtain the final CRC value.

The VHDL code was generated by [55], which is an online tool to generate VHDL code for a parallel CRC algorithm. The data width can be specified as well as the polynomial size and its coefficients. The VHDL code of the algorithm can be seen in Appendix A. Note that the *lfsr_q* is the CRC output and the *lfsr_c* is the next state of the CRC generator. The *crc_en* port enables the generator to update the output register to the next state. A reset port (*rst*) is provided in order to force the internal state to the initial state. The input data is a vector with 32 bit. The clock source for this block is a counter with one bit and a period 16 times greater than the bit period.

Since the input ports must be carefully handled, a control block called *crc_tx_ctrl* implements a state machine that generates all the required control signals. This block is a *MCode* block which can be configured with a MatLab function. Despite a limited number of MatLab commands are accepted by this block, a state machine can be fully implemented in an high-level programming language, which simplifies the process of the system design. The complete MatLab code for the emitter state machine control can be seen in Appendix A. The design of the controller considers three input ports: *data_valid* signals that the data at the input of *CRC_Generator* is valid, *data_length* is the number of input 32 bits data symbols and a *reset* port. For the outputs, were considered four signals: *crc_enable* and *crc_reset* which drives directly the *CRC_Generator* block; the *data_out_select* selects the output data (input data or CRC remainder); finally, the *data_valid* output is a signal that acknowledge the output data validity.

The *Byte_reflect* block after the gateway in performs a conversion from serial bits to bytes. However, the input order considers that the last input bit is the least significant bit of the output byte. This block introduces a delay of 4 bytes. A posterior serial to parallel operation groups 4 (reflected) bytes into a word of 32 bits which is the input of the CRC Generator block, introducing a delay of 32 bit periods. Thus, the data is only valid at the input of CRC Generator after two 32 bit word and a delay must be inserted in the *data_valid* input. At the output of the CRC Generator, a downsample block is used to convert the sample time of previous block which is the same of the clock signal. The next step is to reflect the CRC remainder bits and apply the XOR operation with 0xFFFFFFFF. The output data is then selected between the reflected input data and the CRC output and converted to a serial communication. Along with data output, the signal *Ouput_valid* is sent in order to inform if the data output is valid.

The CRC receiver model is shown in Figure 4.31. The main principle of the receiver is to compute the CRC of received data and compare it to the CRC received value. The inputs of this model are the *Data_input* and *Data_valid* that are produced by the emitter. The *Data_length* and *Reset* are also inputs of this model, as in the emitter model. After conversion of received data to 32 bit words, the CRC remainder are computed in the *CRC_Generator_RX*. The block that controls the CRC Generator is similar to the implemented in the emitter. However, the *data_out_select* was replaced *crc_valid*. This new output validates the comparison between the computed CRC in the receiver and the received CRC remainder.

The source code of the *crc_rx_ctrl* block can be seen in Appendix A.

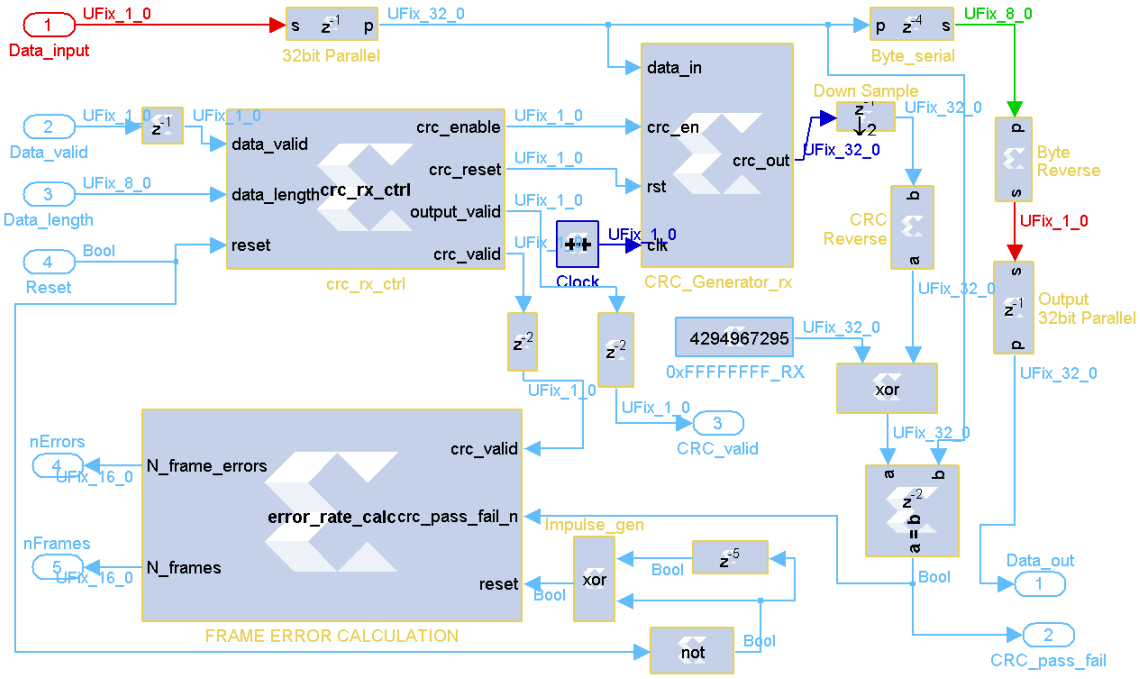

Figure 4.31: System Generator CRC-32 receiver synchronous model

The outputs of the receiver model are *Data_output* and *Data_valid*, which are the data bits output and the signal to validate the data output, respectively. After the CRC calculation, a statistical analysis of the received frames is made in the block *error_rate_calculation*. As in the case of CRC control blocks, this is a *MCode* block and analyses the result of the CRC comparison for each received frame. The state machine implemented in its MatLab function computes the number of received frames and the number of frames that contains error, in other words, the number of received frames that presents a mismatch on the CRC field. The reset input clears the output values in the beginning of the simulation. The full source code for the state machine of this block can be seen in Appendix A.

For model validation purposes, an error probability of $p = 1/(10 \times 212 \times 8)$ was chosen, considering an average of one error bit for each 10 frames of 212 Bytes. As can be seen in the model displays, $FER = 203/2000 = 0.1015$, which is the expected value.

4.2.5 Hardware co-simulation

In order to obtain more accurate results in less time, the System Generator allows to create models using Hardware co-simulation. This System Generator feature makes use of the FPGA hardware to run the model simulation. In other words, it performs the implemented hardware blocks in the FPGA using acquired data from System Generator and returns the processed data. This data transmission occurs via the JTAG port of the Spartan SP605, which has a limited bitrate.

The main goal is to implement as much as it possible the model in hardware blocks in order to minimize the simulation time. Besides, the amount of data exchange between the

System Generator and the FPGA shall be as lower as possible in order to take advantage of the hardware implementation capabilities.

Before going into detail on the ECC co-simulation models, first it will be presented the used methods to reduce data transmission over JTAG port. Despite the model is fully hardware implemented (data generator, error generator, encoder, decoder and error rate calculation), the error probability parameter shall remain selectable in order to perform simulation at different P_e values. On the other hand, at the output, a statistical analysis of error rate shall be delivered to the System Generator environment, particularly the number of received bits and the number of received errors.

Since the error probability is a constant value during the simulation, this parameter is sent to the FPGA at one sample per second, explicit in the gateway in of the models. However for the output values, it is required a down sample operation, reducing the amount of values per time unit that is sent to the System Generator.

In addition, in these hardware co-simulation models was used a structure called *Shared FIFO*. This structure is a common FIFO, however, it is divided in two different blocks: *To FIFO* in which the data is inserted into the buffer and *From FIFO* which is the complementary function of the first block that allows to read the data inside the buffer. Using this structure, the FPGA clock and the System Generator sample time can be completely different, allowing the hardware implementation to run at the FPGA clock speed while the System Generator only takes few values from the *From FIFO* block at a much slower rate. Both word length (in bits) and FIFO depth (maximum number of stored words) can be configured in these blocks.

Bellow are described the co-simulation models for RS and Convolutional codes. The taken approach in these models is to generate both data and errors as well as implement the *Additive Noise Channel* with FPGA blocks. The first model implements only the BSC in order to give to the reader an idea of how the Simulink blocks were replaced by Xilinx blocks. The Figure 4.32 is shown the co-simulation model for the BSC.

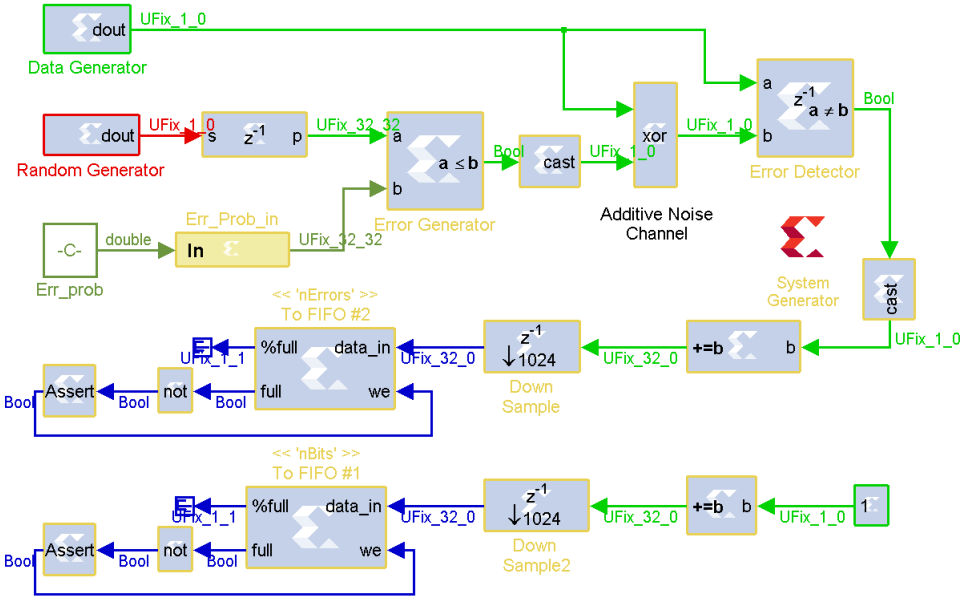

Figure 4.32: BSC co-simulation System Generator model

The block *Data Generator* is a LFSR with 32 bits that generates a pseudo-random sequence. The parameters of this block are:

- Type: Galois
- Gate type: XNOR
- Number of bits in LFSR: 32
- Feedback polynomial: 0x80200003
- Initial value: 0x55555555

The feedback polynomial was chosen in order to provide the longest pseudo-random sequence, according to [56]. The sample rate of this block is 32 times slower than the minimum period of the FPGA (20ns).

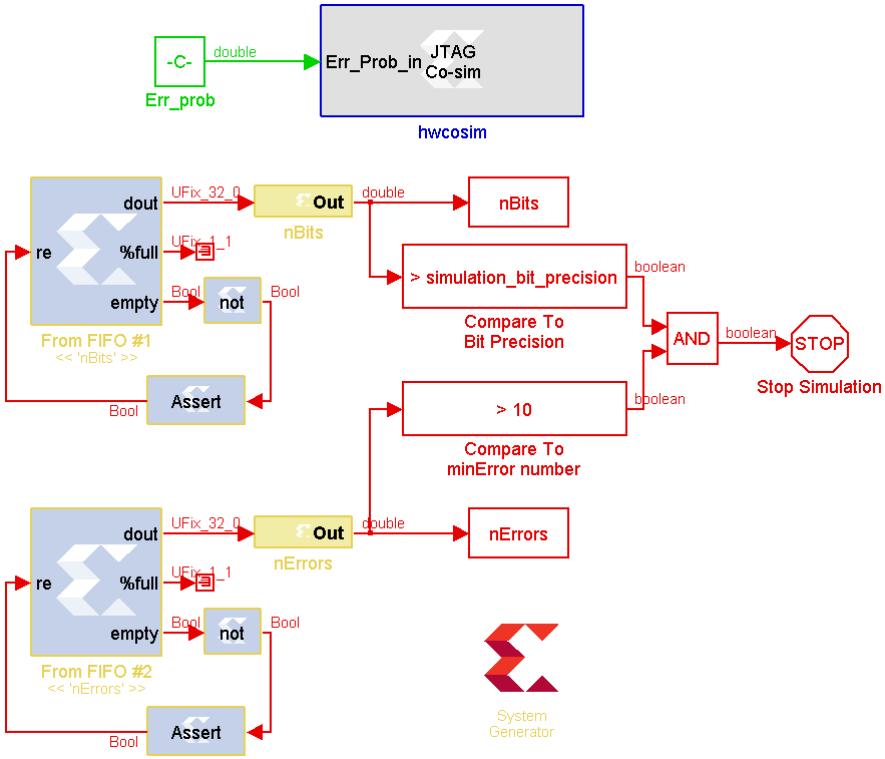

Figure 4.33: Generic model for hardware co-simulation data acquisition

Identical to the previous block, the *Random Generator* is also a LFSR. However, the sample rate is 20ns which is the minimum allowed for hardware co-simulation in the Spartan 6 SP605 Evaluation Kit. The output random bits are then grouped in 32 bits parallel and posteriorly compared to the error probability. Note that the Gateway In for the error probability interprets this value as a 32 bit fractional number. The *Error Generator* block sets its output high if the word randomly generated is lower than the error probability constant.

The blocks described in this paragraph implements a Bernoulli generator, identical to the previously used from Simulink block set.

The Additive Noise Channel is also implemented in hardware with the XOR function. Since the output type of the *Error Generator* block is Boolean, a conversion is made before the XOR operation between Data bits and Error bits. The result is then compared to the original data and an accumulator block is used to count the number of errors in the channel. Parallel to error counter, an additional accumulator block counts the number of transmitted bits, which in this case is merely a constant with value=1. The output of each accumulator is down sampled in order to decrease the amount of data that will be sent to System Generator environment using the *Shared FIFO* structure, as previously explained.

In order to receive the data from the hardware, it was used the model shown in Figure 4.33. As the reader can observe, the *From FIFO* blocks have the same name of the *To FIFO* blocks in the Figure 4.32, which means that they are part of the same FIFO structure. The read sample time from these blocks is 1024 times slower than the bit sample time in the main model, allowing the System Generator to work slower than the FPGA. The output data is sent through the gateway outs and are stored in MatLab variables (nBits and nErrors). Furthermore, the simulation stops if both conditions are verified: the number of simulated bits is greater than the desired value (*simulation_bit_precision*) and the number of received errors is greater than 10, in order to have a proper value to compute BER.

The measured error probability was according to the expected value, as Figure 4.34 shows.

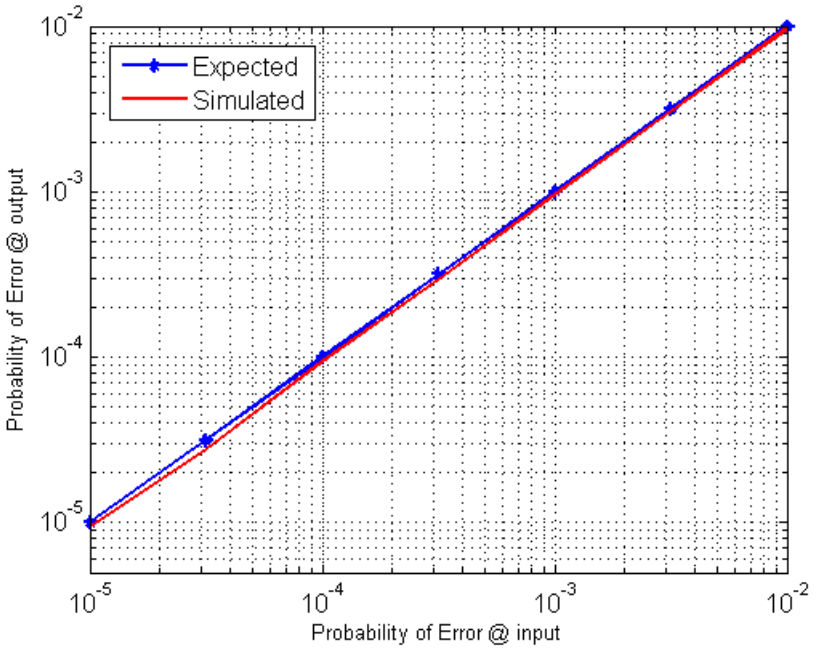

Figure 4.34: Hardware co-simulation results for the a BSC

With the previous model in mind, both Reed-Solomon and Convolutional System Generator models were modified in order to perform fast simulation using a full hardware implementation. In Figure 4.35 is shown the co-simulation model for the RS(255,213) code. Note that the error rate calculation section compares the input symbols from emitter and receiver

only when the *output_valid* at the receiver is set.

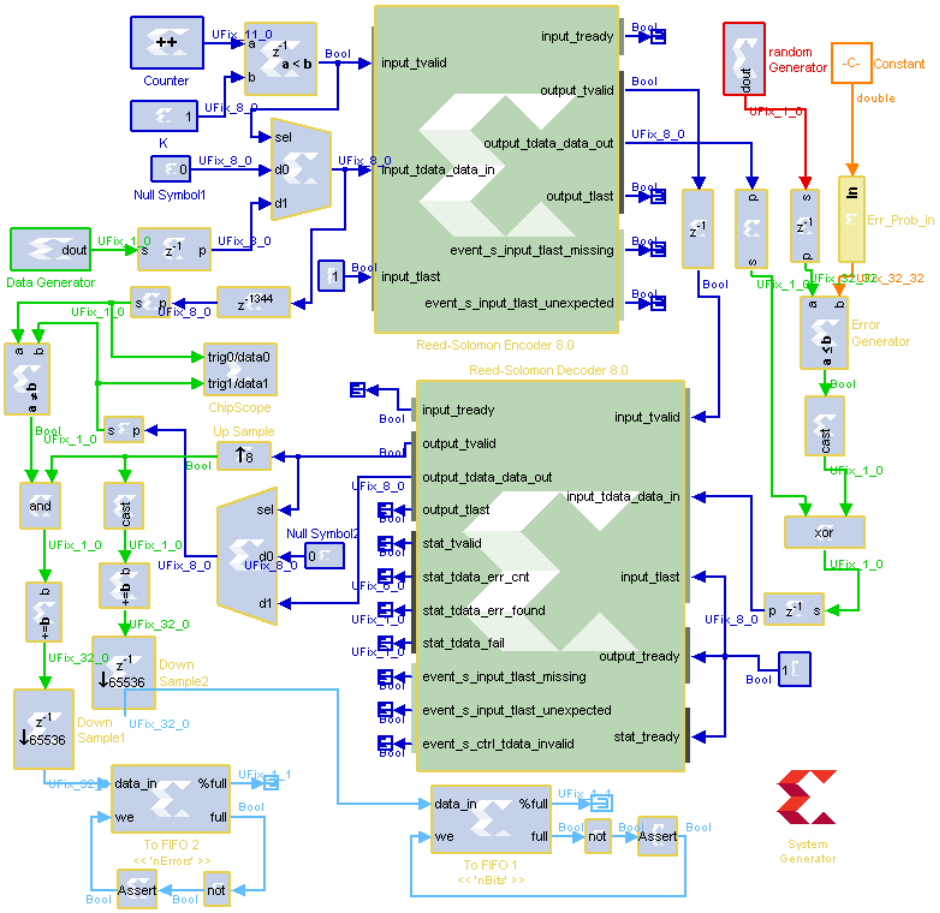

Figure 4.35: RS(255,213) co-simulation System Generator model

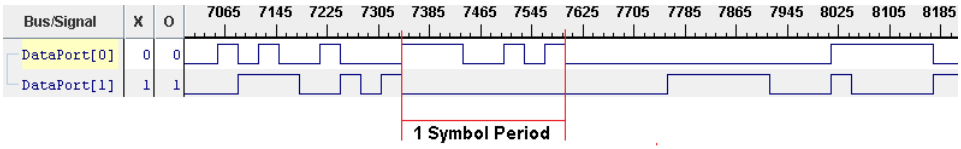

Figure 4.36: ChipScope Pro waveforms of input and input data bits, showing an misalignment between signals

An important remark in the latency between input and output symbols shall be made. In the first approach, the same latency as in the model of the Figure 4.22 was considered. However, after perform the co-simulation, the achieved BER was approximately 0.5. This value suggests a misalignment of input and output bits at the Relational block input. In order to observe the internal FPGA signals, a ChipScope block was inserted in the model to acquire data of the Relational block inputs. The Figure 4.36 shows the misalignment between the signals which presents an unnecessary latency of one symbol period in the input

data signal. Thus, the total delay value shall be 1344 symbol periods instead of the previous value of 1345 periods. The same approach was made for the Convolutional codes, as seen in Figure 4.37.

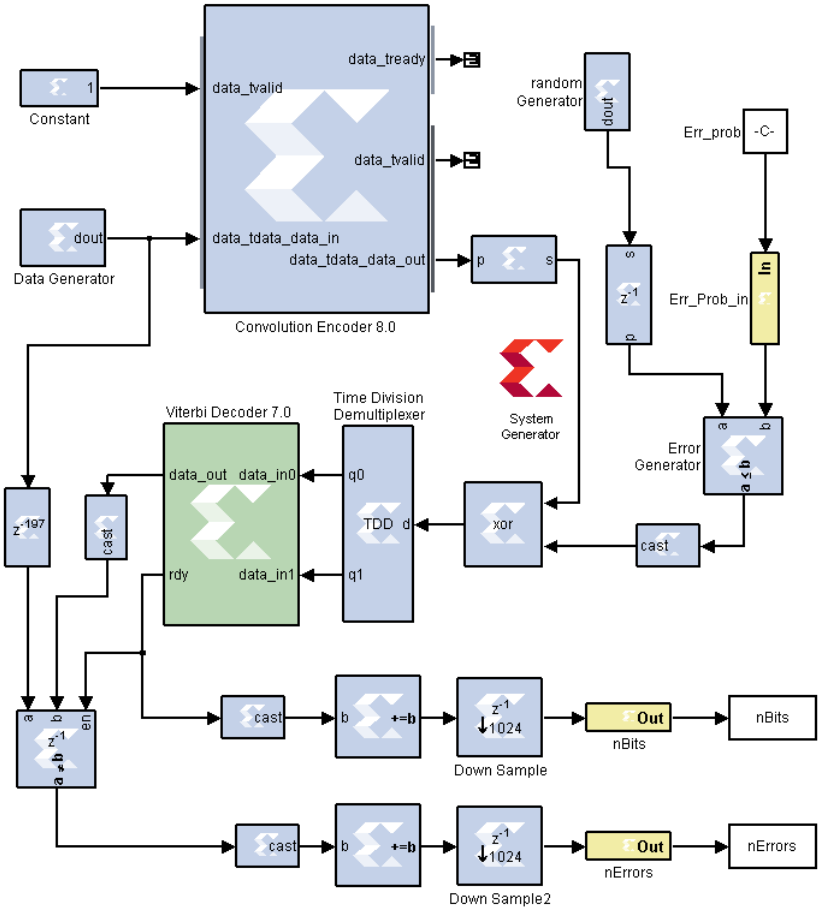

Figure 4.37: Convolutional code R=1/2, K=7 co-simulation System Generator model

The results obtained in hardware co-simulation for Reed-Solomon and Convolutional Codes are shown in Figure 4.38 and 4.39, respectively.

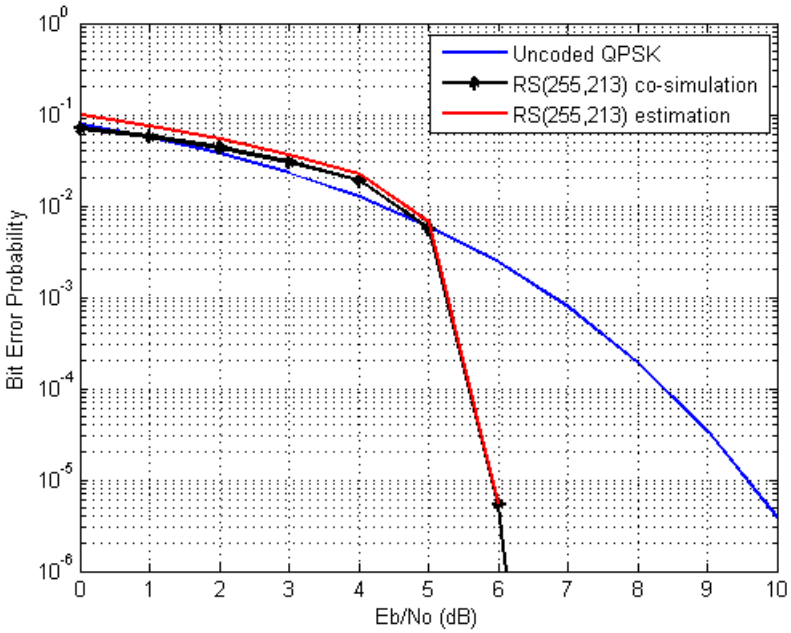

Figure 4.38: Performance of the RS(255,213) co-simulation System Generator model

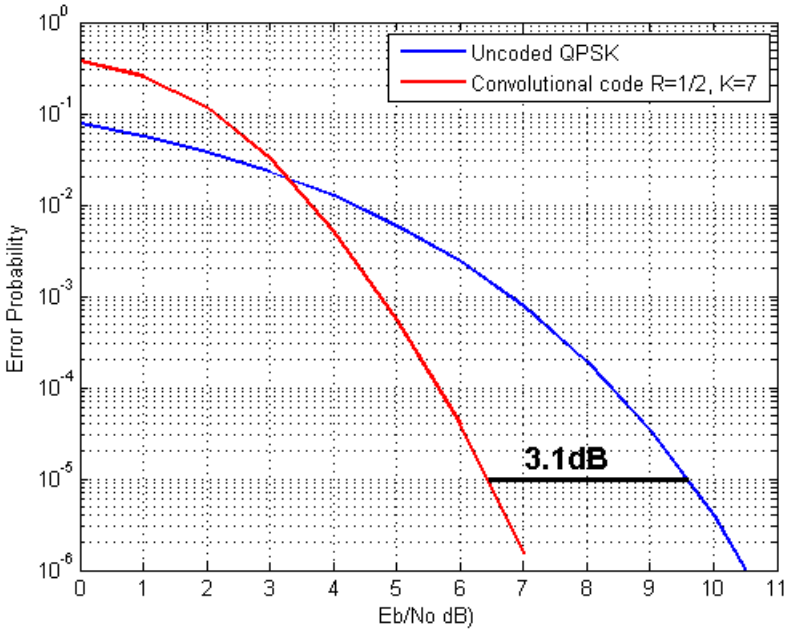

Figure 4.39: Performance of the Convolutional code R=1/2, K=7 co-simulation System Generator model

4.3 System Constraints

As previously said, this dissertation study focuses in the implementation of an effective FEC scheme for a VLC system. Therefore, the choice of a specific scheme must consider the system parameters, in particular the DLL and the modulation scheme.

The main goal of the encoder in a communication system is to encode the data to be transmitted in order to improve the BER of the system. In this project, two data services were considered: a Moderate Data Rate (MDR) used for control and management as well as for advertising and infotainment services; A High Data Rate (HDR) for video broadcast. Since these services must be transmitted through the same physical channel, the DLL must process the requests of data transmission from higher layers. Therefore, this layer sends in the same frame both services, giving priority to the MDR. Furthermore, if a transmitting request length is larger than the supported transmitting length, the input data shall be fragmented and sent in multiple frames which are reconstructed in the receiver side. The frame structure of the DLL is shown in Figure 4.40.

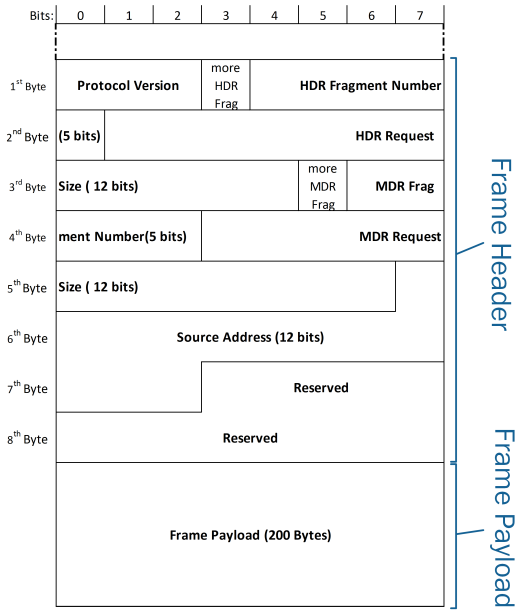

Figure 4.40: Frame structure

As it can be seen, the frame header comprises 8 bytes. Before going into detail on each header field, an important remark shall be made about the frame payload. This field contains MDR, HDR or both services, depending on the transmitting requests from the higher layers. In the case of the occurrence requests from both services, the DLL allocates fixed sizes for each service, prioritizing the HDR service. Therefore, the frame size must be chosen carefully in order to minimize the fragmentation of the HDR service [2]

One of the most common used container formats for transmission and storage of audio and video is the MPEG transport stream (MPEG-TS). Since it is used in video broadcasting systems, such as in DVB, the MPEG-TS standard shall be considered in the DLL frame size choice [57].

The models described in previous sections are an important tool to evaluate the performance of different FEC techniques that will be later discussed in Chapter 5. At this point is important to stablish which scheme will be used in the final FEC implementation, based on each ECC performance. In the Figure 4.41 is a representation of the system encoder and decoder.

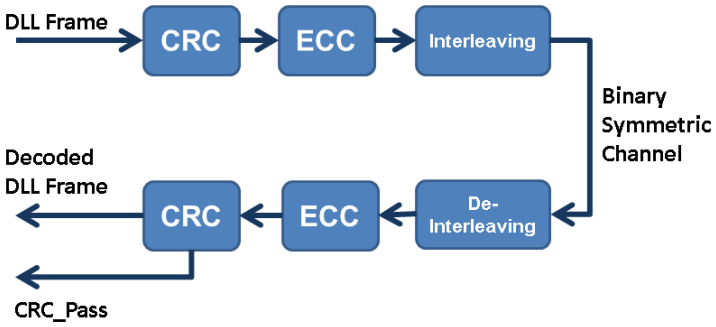

Figure 4.41: VLCLighting encoder and decoder schemes

The first observation is that a CRC-N detection scheme shall be used in order to enable the received decoded data validation. This block signals the DLL if the received frame contains remaining errors that could not be corrected by the FEC techniques. In order to provide a reasonable detection capability to the CRC algorithm, the CRC-32 standard was considered, providing a detection probability of 0.99999999977, according to equation 3.25.

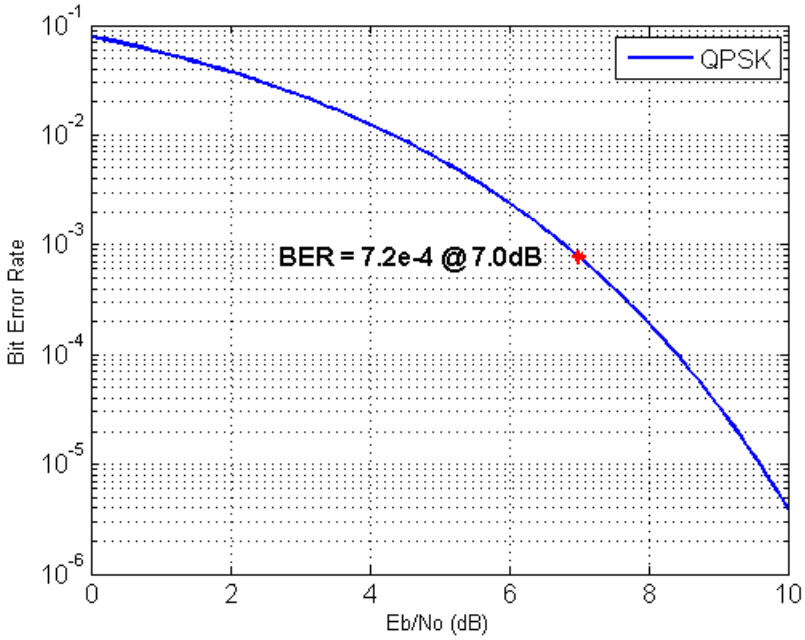

Figure 4.42: Performance of QPSK modulation over an AWGN channel

According to the last results of VLCLighting project in [58], for a single LED system, the BER of received data is at maximum 10^{-3} for a distance of 1.75m. This results were achieved with an LED that consumes less than 1W. For outdoor scenarios is expected an higher luminous output, providing an higher signal power. In other words, in the final stage of the project the available optical signal power will be higher that will improve the SNR and consequently reduce the BER. Therefore, the BER results for the 2.0m scenario was not considered in this study.

The modulation scheme used in VLCLighting is a combination of QPSK and OFDM, more specifically ACO-OFDM. As mentioned in Section 3.3, the OFDM modulation along with a proper interleaving technique shall mitigate burst errors and a BSC must be considered in the study of ECC. First of all is important to characterize the channel in terms of E_b/N_0 and BER. The Figure 4.42 shows the BER performance of a QPSK modulation in a narrow band AWGN channel.

The results in this chapter for the ECC performance were obtained by using the *bercoding* and *berawgn* MatLab functions. This function computes an estimative of the decoding BER for a particular code used along with a specified modulation scheme as well as the E_b/N_0. The MatLab script that was used to obtain the results in this section can be seen in Appendix B.

The equivalent E_b/N_0 of the BER obtained in [58] is about 7dB. This value will be considered the maximum allowed value in order to proper system functioning.

A comparison between a Convolutional code with R=1/2,K=7 and a RS(255,213) performance is shown in Figure 4.43.

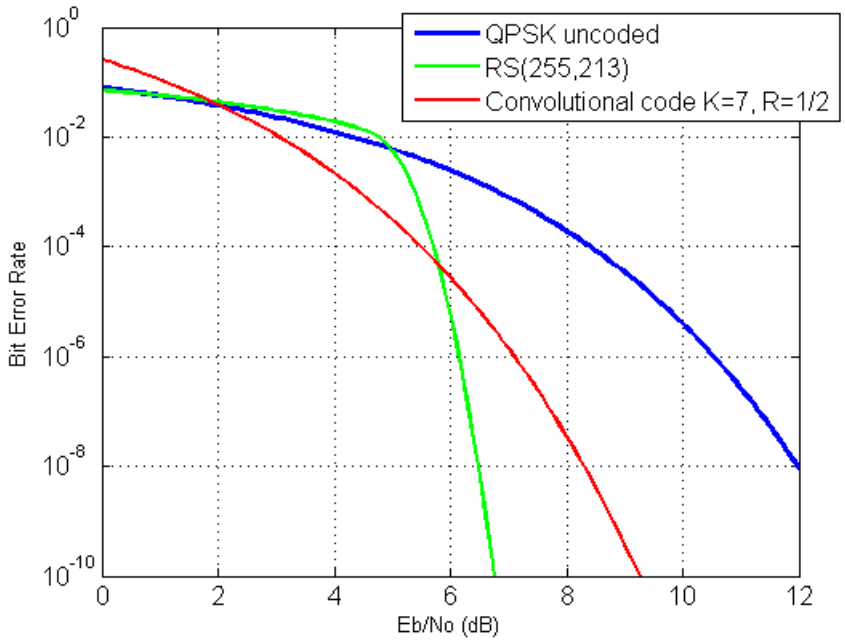

Figure 4.43: Performance of RS(255,213) and Convolutional (R=1/2, K=7)

As the reader can observe in the figure, above 5.8dB of E_b/N_0 the RS codes outperforms the Convolutional code. Furthermore, the efficiency of the usedRS(255,213) code is 83.53%, which is higher than the Convolutional code efficiency, 50%. Therefore, the RS codes shall be

preferred as opposed to Convolutional codes in order to accomplish both system performance and efficiency. Additionally, the RS(255,K) achieve a better match with Byte addressable architecture of the DLL. After selecting the most appropriate ECC, a study of RS(255,K) codes shall be considered in order to observe the evolution of error correction capability by varying the number of data symbols. The results of this analysis can be seen in Figure 4.44.

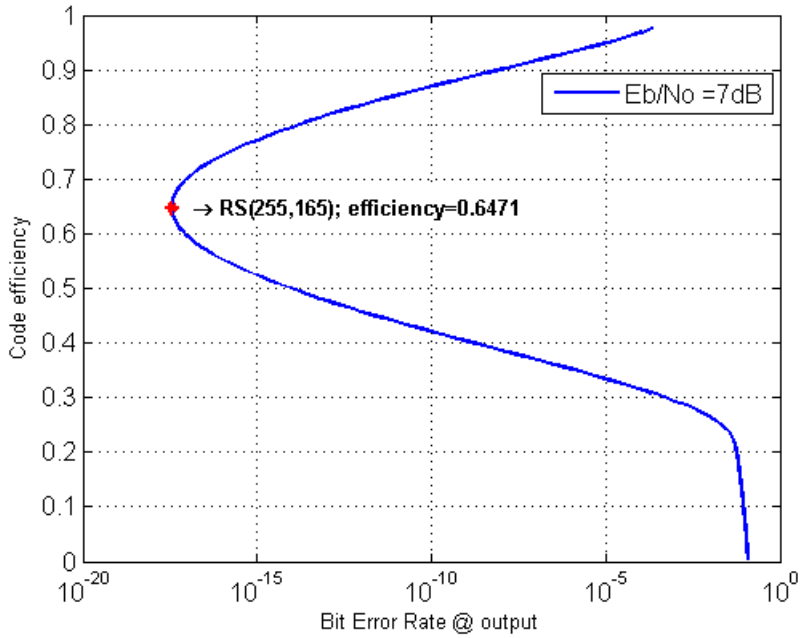

Figure 4.44: Reed-Solomon(255,K) codes efficiency vs BER performance

As opposed to the common thinking, the results shows that the performance of the code starts to degrade if more than $(255-165) = 90$ correcting symbols are used. This is due to the signal power normalization: the transmitting power is the same for the uncoded transmission (K symbols) and the coded transmission (N symbols). This effect can be seen in equations 3.18 which is a parameter of RS symbol error probability (equation 3.16), where the E_b/N_0 value is multiplied by R. In the particular case of RS(255,K) code family the minimum K worth using is 165 which gives a code efficiency of 64.71%. This value provides a very strong correction capability and it is once again above the efficiency and performance of Convolutional codes of R=0.5. Thus, the choice of RS(255,K) codes instead of Convolutional codes was the right one.

The next step is to compute an adequate value of parameter K of the code. Since the frame size can require more than one codeword to encode it, the definition of Block Error Rate (BLER) and Frame Error Rate (FER) must be established. The parameter BLER is related to the error probability of an existing wrong symbol in a codeword and is given by:

$$BLER = 1 - (1 - BER)^K \tag{4.5}$$

In the other hand, FER is the probability of a received frame with errors. In this context, this parameter means the probability of an existing codeword with errors in the frame and is

given by:

$$FER = 1 - (1 - BLER)^{N_{BLOCKS}} \tag{4.6}$$

A desired FER can be easily calculated using equation 4.7.

$$FER = \frac{1}{Frames\_per\_second \times time\_without\_errors} \tag{4.7}$$

where $Frames\_per\_second = bitrate/(8 \times frame\_size)$ and $frame\_size$ is expressed in Bytes.

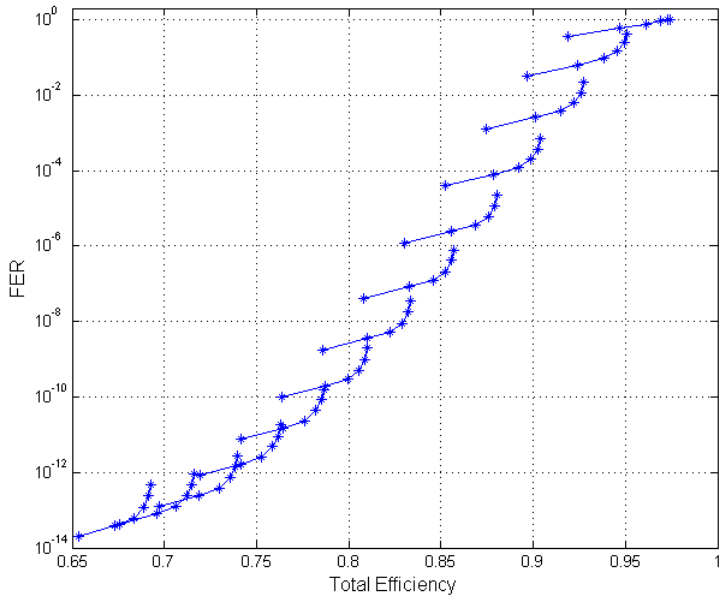

Figure 4.45: FER vs system efficiency

At this point it is interesting to understand how the FER behaves for different values of K and the impact in the total system efficiency ($frame\_efficiency \times code\_efficiency$). This relation can be observed in Figure 4.45 where each line represents one RS(255,K) and each point of the line is a different frame size (128, 256, 512, 1024, 2048, 4096). Note that to better understanding of this figure, not all RS codes were included. Starting from RS(255,165), steps of 6 in K value were considered. It is observed a superposition of several values in the plot. It is also concluded that for the same ECC, a larger frame results in an higher FER value, however with an higher efficiency.

It was considered that the application of VLCLighting must provide a continuous operation without errors during one hour. Thus, the FER value for a 24Mbit/s [58] and a frame size of 256 Bytes is 2.3704×10^{-8}. The frame size 256 is a first approach value which includes the 8 Bytes from frame header, the 188 Bytes from the MPEG-TS frame, 4 Bytes from the CRC-32 remainder and the remaining Bytes comprises the MDR data. This last parameter of the frame is dependent of how many Bytes remain in the codeword data field after a chosen K.

In Figure 4.46 is shown the FER values and respective system efficiency values for RS codes from K=165 to K=253, considering a frame size+CRC of 256 Bytes. Note that this results are a particular set of the previous obtained values. The horizontal line in the graph establish the limit for the previous calculated FER value. Therefore, all RS(255,K) codes below this line are eligible to fulfil the system requirements. However, the code with the highest possible K shall be chosen in order to provide both system efficiency and the higher MDR possible. The RS(255,215), highlighted in the figure, meets the FEC specification with an efficiency of 79.61%.

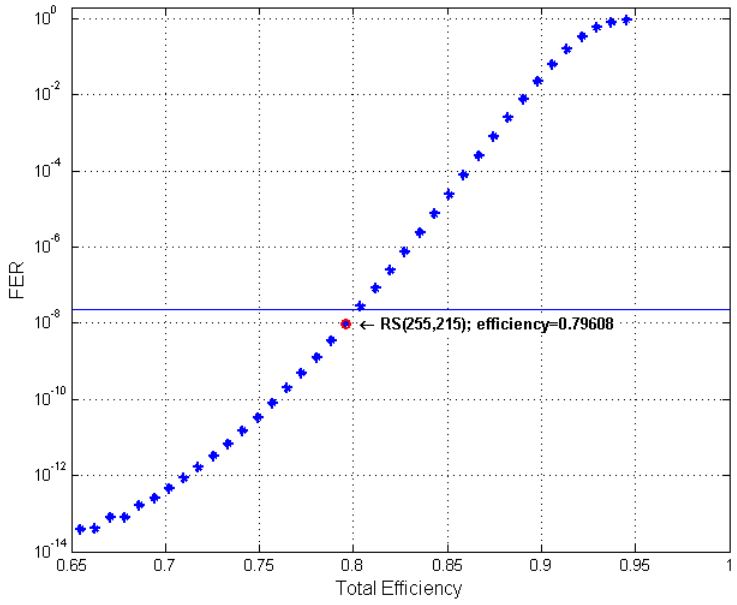

Figure 4.46: FER vs system efficiency for a frame size=K-CRC

Due to the constraints of the DMA used in the DLL, addresses and transfer size need to be aligned to 8 Bytes. Therefore the previous considered frame size of $215 - 4 = 211$ shall be aligned to the immediately lower possible value, which is 208 Bytes. With the additional 4 Bytes of the CRC-32 remainder, a total of 212 Bytes will be transmitted. One last adjustment of the code parameters must consider RS correction capability. According the equation 3.14, and in order to correct an integer number of symbols, $N - K$ needs to be even, thus, K must be odd. This condition implies that a junk symbol must be inserted after the CRC-32 remainder. Thus, the code to be implemented in the project is a RS(255,213), which can detect up to 42 wrong symbols and corrects up to 21 symbols. From equations 4.5 and 4.6 the expected FER at 7dB of E_b/N_0 is 2.196210^{-10}. Furthermore, using this code, the system efficiency is given by:

$$\eta = \frac{K - Junk\_symbol - CRC - Header\_size}{N} = \frac{213 - 1 - 4 - 8}{255} = \frac{200}{255} = 78.43\% \quad (4.8)$$

A last remark on the code choice is that the available number of Bytes for the MDR

service is $200 - 188 = 12$, which is 6% of the total payload size. Note that this restriction is only applied when the DLL receives requests from both data services.

4.4 Concluding Remarks

In this chapter were presented Simulink and System Generator models for several FEC techniques. The results of these models were compared with the theoretical values. This process helps to better understand each FEC characteristics.

The hardware-cosimulation tool proved to be useful. When compared with the same model without co-simulation, the simulation time can decrease from several hours to few seconds.

At the end of this chapter, it was concluded that a combination of CRC-, RS(255,213) and a block interleaver would be the better Channel Encoder approach to the considered scenario.

Chapter 5

Implementation and Final Results

5.1 Channel Encoder model

As previously said, the proposed Channel Coding scheme comprises three functionalities: CRC-32 algorithm in order to provide data validation, RS(255,213) codes in order to perform error correction in the decoder and an Interleaving technique to improve RS capabilities in the presence of burst errors.

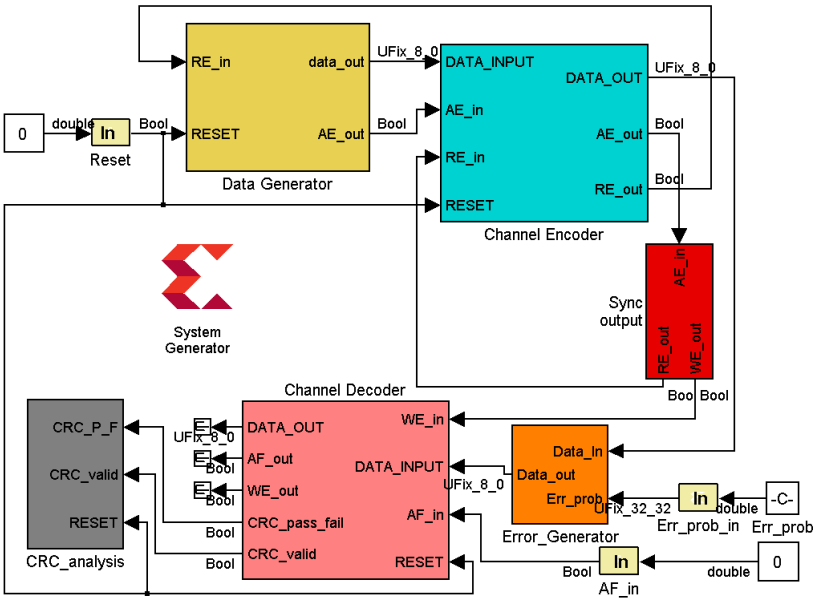

Figure 5.1: Asynchronous architecture of the proposed Channel Coding scheme

In Figure 5.1 is shown the System Generator model of the proposed Channel Coding scheme, implemented in an asynchronous architecture. As the reader can observe, it has 6 blocks which will be described in detail in this section: Data Generator, Channel Encoder, Error Generator, Sync Output, Channel Decoder and CRC Analysis. The implementation considers an 8 bits data width between blocks operating at the Spartan 6 frequency (50MHz).

The Data Generator block generates the data to be transmitted. As it can be seen in

Figure 5.2, a Read-only-Memory (ROM) is used to store the Bytes to be sent. The ROM address value is given by a Counter block which are enabled/disabled by the TX_BUFFER_-CTRL block and the the output values are stored in the output FIFO of the asynchronous architecture. Note that the FIFO Write Enable (WE) signal is delayed by one period in order to match the ROM latency. The FIFO level state is monitored by the signal "%full" which is given with 4 bit precision and posteriorly converted from fractional to integer notation. A RESET signal, which resets the TX_BUFFER_CTRL state machine and consequently the Counter and the output FIFO. The Read Enable (RE) signal is controlled by the Channel Encoder block and reads out the FIFO values to the data_out output port. Additionally, an Almost Empty signal is sent to the Channel Encoder in order to guarantee that the output FIFO has enough stored valid data.

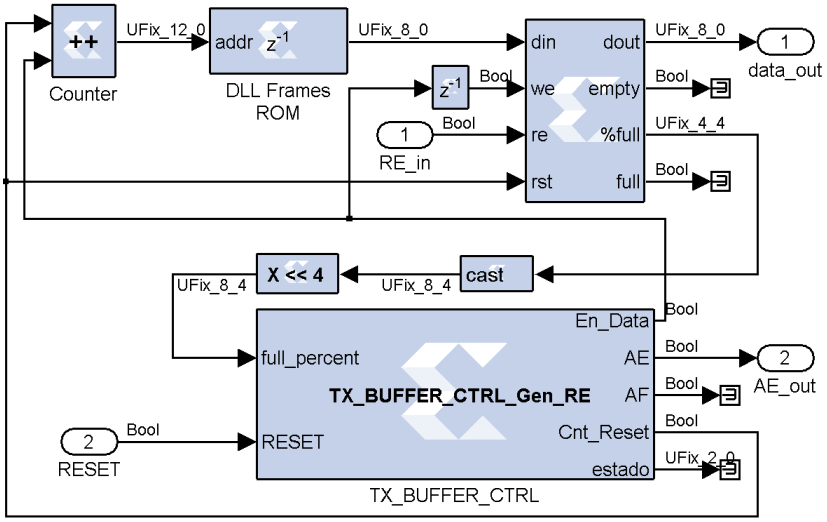

Figure 5.2: Asynchronous architecture - Data Generator

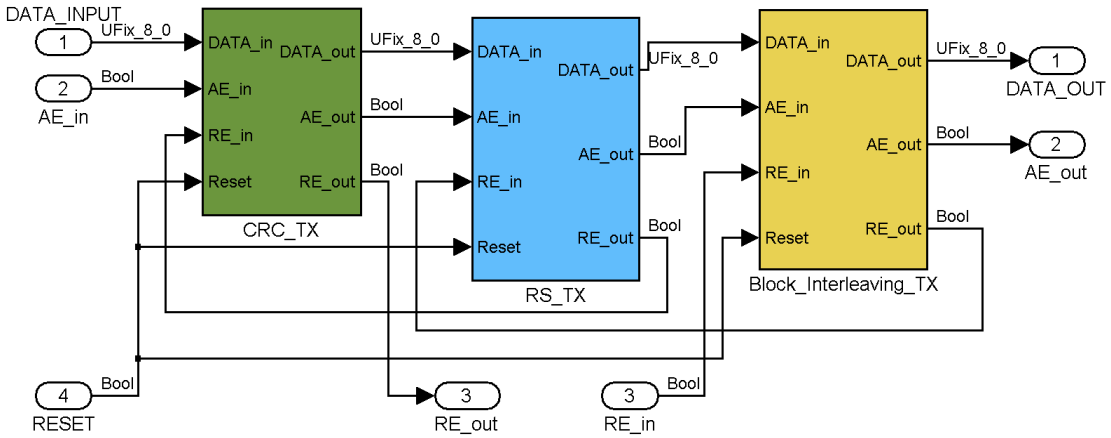

Figure 5.3: Asynchronous architecture - Channel Encoder

The following block is the Channel Encoder, represented in Figure 5.3, and is formed by three subsystems: CRC_TX, RS_TX and Block_Interleaving_TX. Each block is implemented according to the asynchronous architecture, providing an AE signal to the following block along with the RE signal to the previous block. The input data feeds the CRC_TX block along with the AE signal from Data Generator. The CRC_TX is also responsible for control the read operation from the Data Generator. At the output, the encoded data is provided by the Block_Interleaving_TX which also returns its FIFO lower level state. The signal RE from the following block controls the read operation of Block_Interleaving_TX internal output FIFO. Now that the reader has an idea of how the Asynchronous Architecture works and how the Channel Encoder is structured, a detailed description of the each block shall be made.

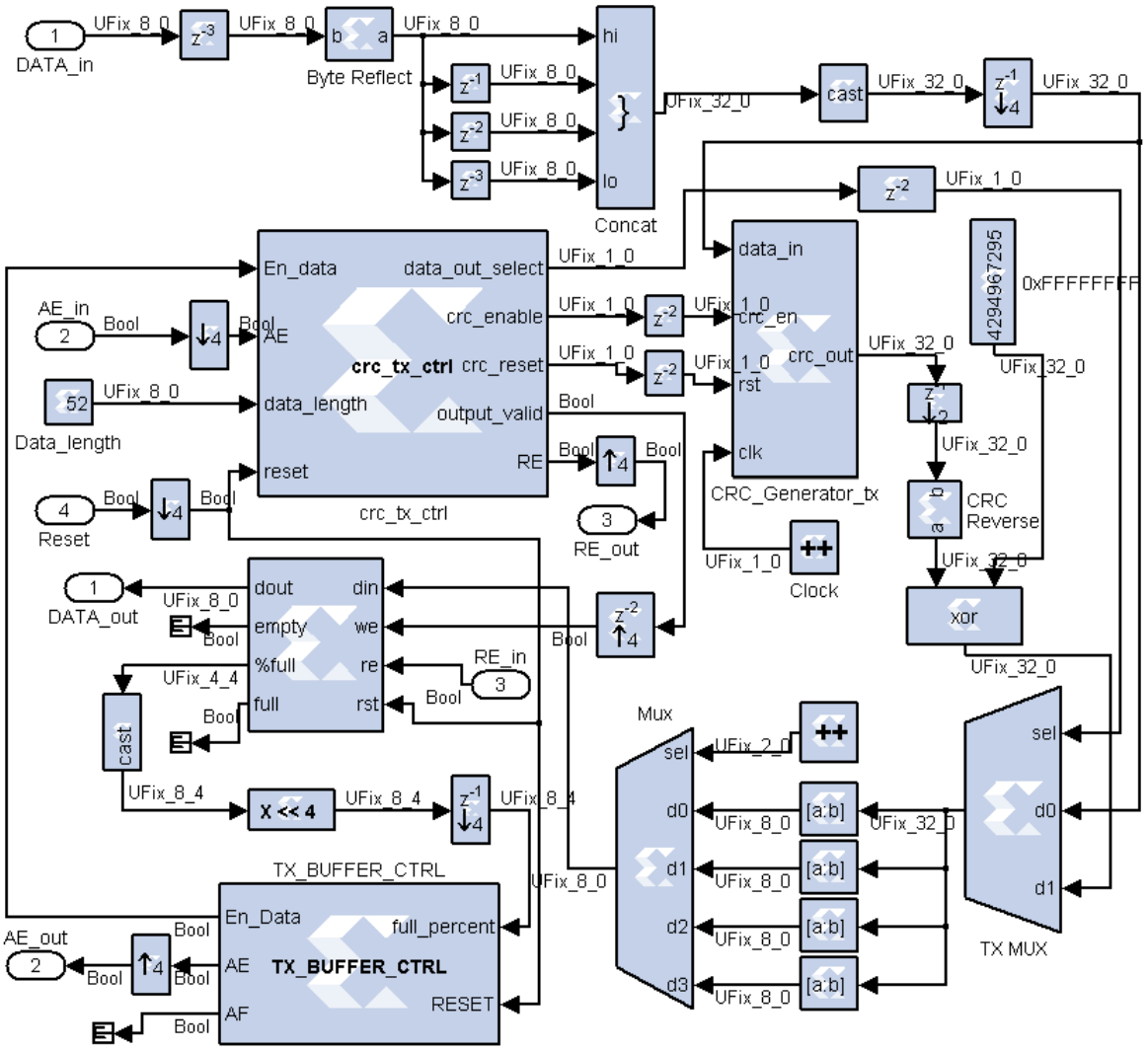

Figure 5.4: Asynchronous architecture - CRC-32 encoder

In Figure 5.4 is shown the CRC-32 asynchronous encoder. As in the synchronous model, a CRC generator block controlled by a Finite State Machine (FSM) is used. However, some changes were made in order to provide a processing speed of one byte per clock cycle.

The implementation of the CRC FSM considers four states: RESET, DATA_READ, CRC_RESULT and WAIT. The RESET state initializes all the outputs and variables. The DATA_READ state is responsible to read all the 208 Bytes (52 words) from the input data that is followed by the CRC_RESULT state that outputs the CRC-32 value. The FSM enters in the WAIT state either if the AE signal of previous block is set or if the output FIFO is AF (which disables En_Data signal), returning only to the DATA_READ state if both signals have zero value.

Since the CRC generator processes 32 bits each time, its period, as well as the crc_tx_ctrl period, is four times longer than the Byte period. In order to match the periods of the control block with other blocks in the system, a series of downsample and upsample blocks were used. Furthermore, one read operation from the previous block results in an input of four consecutive Bytes.

In the data input, a delay of three byte periods was introduced in order to perform a time alignment between the four input bytes and the period of the processing blocks. Note that the input bytes are already affected by one period of latency due to the FIFO read operation. The Byte is reflected by a Bit Basher block that rearrange the bits in a reverse order. The next operation concatenates four consecutive Bytes in a 32 bits word, giving to the first Byte the lower significant position and the last Byte the highest significant position. After this operation, a downsample is performed to match the CRC generator sample time.

The control signals from crc_tx_ctrl are delayed for two word periods (8 clock cycles) in order to compensate the latency introduced by: 1) the FIFO delay plus the first delay; 2) the delay introduced by the downsample block. Once again to perform fast bit reversion, the CRC bits are reversed using the Bit Basher block. At the output of the CRC-32 encoder, the 32 bits word are converted again into four serial Bytes using Slices (to extract each Byte) and a multiplexer selected by a 2 bit counter. The Bytes are stored in the output FIFO which is controlled by the block TX_BUFFER_CTRL. This FIFO has 512 Bytes storage capacity. The AF and AE thresholds was setted to 81.25% and 6.25% respectively, which correspond to 416 and 32 Bytes respectively. By using these values the FIFO takes into account the existent latencies between the start/stop of processing and valid data output.

The following block of the Channel Encoder is the Reed-Solomon Encoder which are represented in Figure 5.5. It makes use of the RS encoder used in the synchronous model controlled by the RS_CTRL FSM. The processing FSM has four states. The first state is RESET, which initializes all the outputs and variables. If input data is available, the FSM jumps to RS_READ state and starts to read Bytes from CRC-32 block signalling the RS Encoder block that it has valid data at its input port. The delays present in the RS_data_valid and RS_data_last are delayed in order to compensate the FIFO latency. When the last Byte of the total 213 Bytes is being processed, the controller enters in the RS_WAIT state which waits a certain amount of time before starts processing a new codeword. This time, already discussed in Section 4.2.1, is 1074 periods in order to avoid data loss due to the processing delay of RS Decoder. After this time, the FSM enters in a WAIT state, waiting for new input data. Furthermore, if in the middle of RS_READ state output buffer gets almost full or the previous block signals that its buffer is almost empty, the FSM jumps to the WAIT state until the previous conditions are cleared.

The output section of RS Encodes Asynchronous model is identical to the used in the CRC-32 encoder and the same approach concerning to the AF and AE thresholds was taken. Particularly, this block generates more data at the output comparatively to the input. Thus, is extremely crucial to guarantee free FIFO positions to handle with the extra 42 Bytes at

the output.

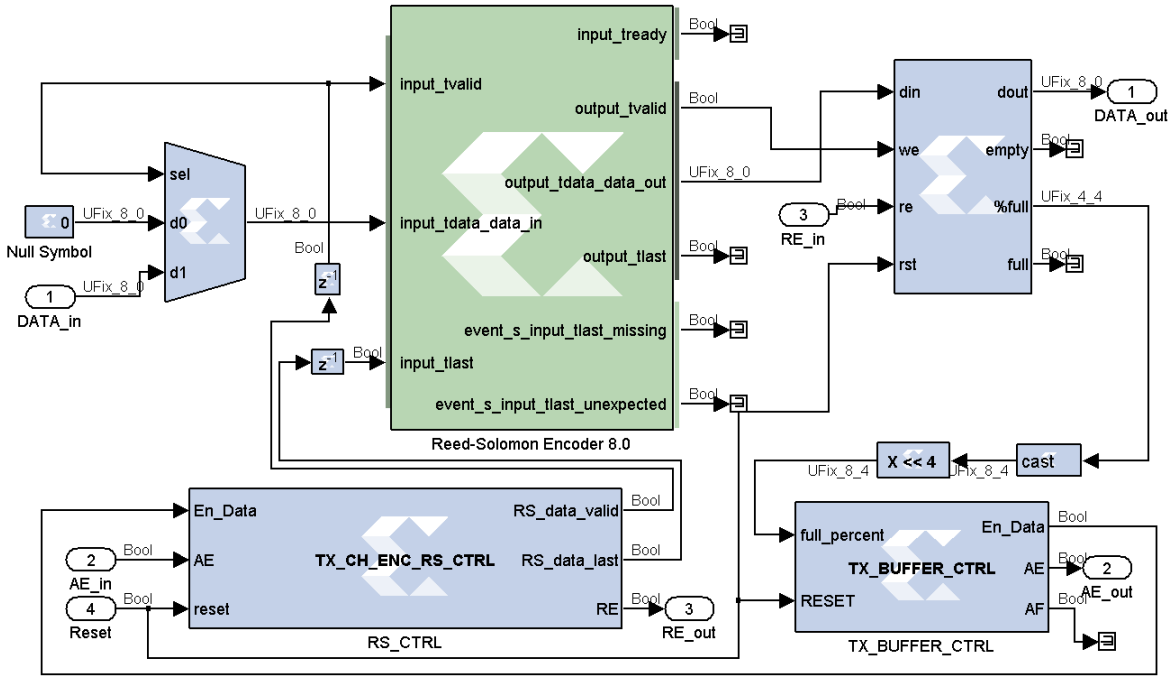

Figure 5.5: Asynchronous architecture - RS encoder

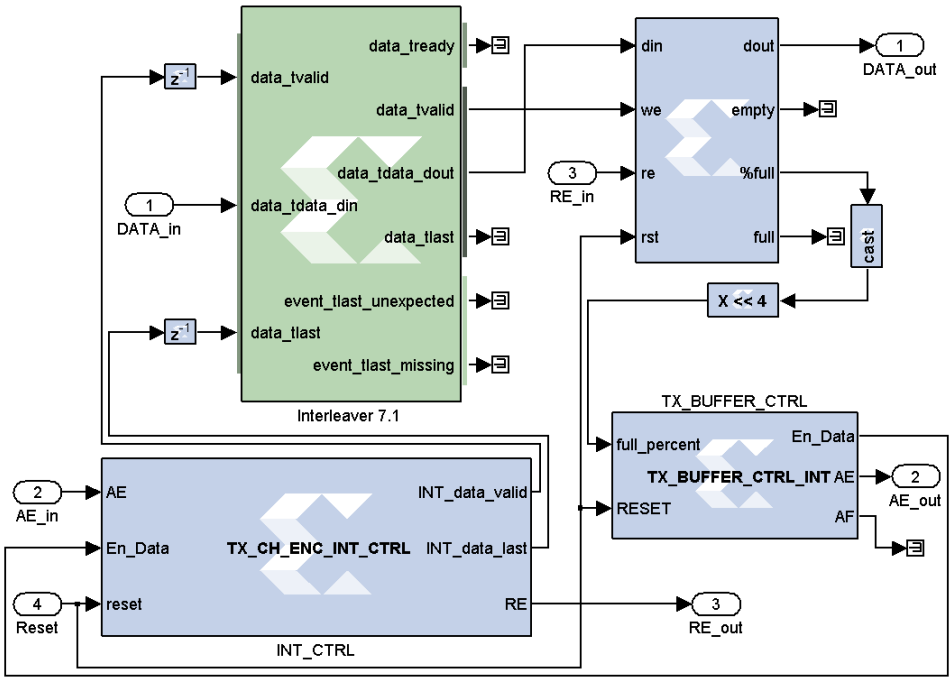

Figure 5.6: Asynchronous architecture - Block Interleaver

The last stage of the Channel Encoder is the Block Interleaving. As in the synchronous model, 10 RS codewords are Interleaved before data transmission. This scheme is very similar to the RS encoder. However, the size of output FIFO is 8K Bytes in order to provide enough storage for 3 complete Interleaver matrix (30 RS codewords). Furthermore, the AF threshold was also adjusted to guarantee that the Interleaver processor can output all the matrix at once without fill, or worse, overflow the FIFO capacity. Hence, if the FIFO reach 62,5% of its capacity (3072 free Bytes) the processing unit is stopped. In other words, the last Byte, which triggers the data output, is only read into the Interleaving processor if the output FIFO is capable of store a complete Interleaving matrix (2550 Bytes).

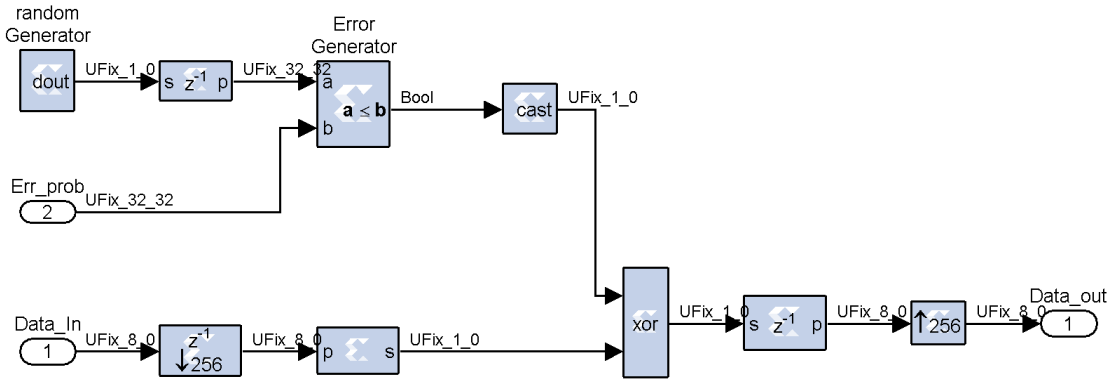

Figure 5.7: Asynchronous architecture - Error Generator

In order to evaluate the system performance an Error Generator was inserted in between the Channel Encoder and the Channel Decoder. The model, shown in Figure 5.7, has the same principle as of the previous implemented Error Generator used in hardware co-simulation. However, since the Relational block named "Error Generator" requires 32 bits from Random Generator block, the data input must be downsampled 32 times. Additionally, this block generates bit errors which means that, for each 32 clock cycles one error bit is generated. Therefore, the total time required to process one input Byte, $8 \times 32 = 256$ clock cycles.

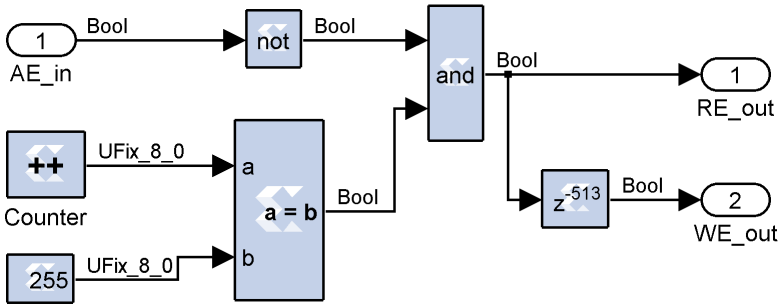

Figure 5.8: Asynchronous architecture - Sync Output

The synchronization between Channel Encoder and Channel Decoder is provided by the Sync Output block, represented in Figure 5.8. Bearing in mind the constraints of Error Generator, if the output FIFO has available data, each Byte will be read to the channel every

256 clock cycles. The WE signal to the Channel Decoder is delayed by 513 clock cycles: 512 cycles from the Error Generator latency and an additional cycle from the output FIFO read latency.

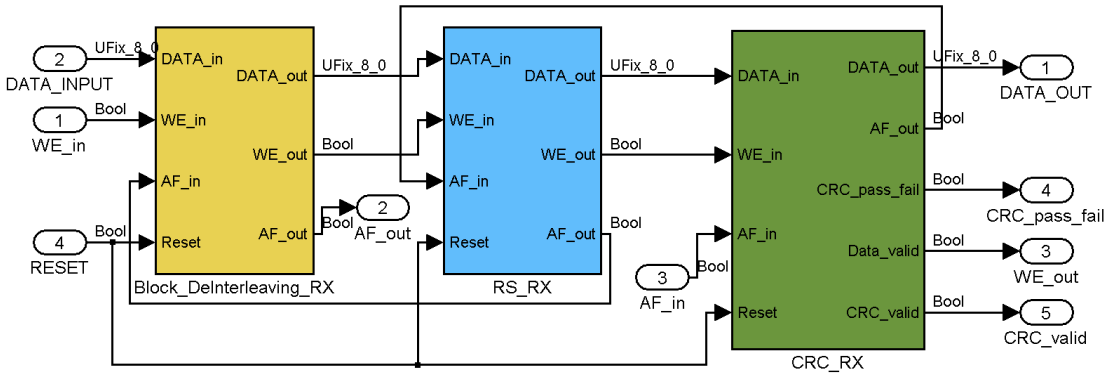

Figure 5.9: Asynchronous architecture - Channel Decoder

The asynchronous architecture was also adopted in the Channel Decoder implementation as seen in Figure 5.9. It decodes the information starting from Deinterleaver process, continuing to the RS decoder and finally to the CRC-32 check.

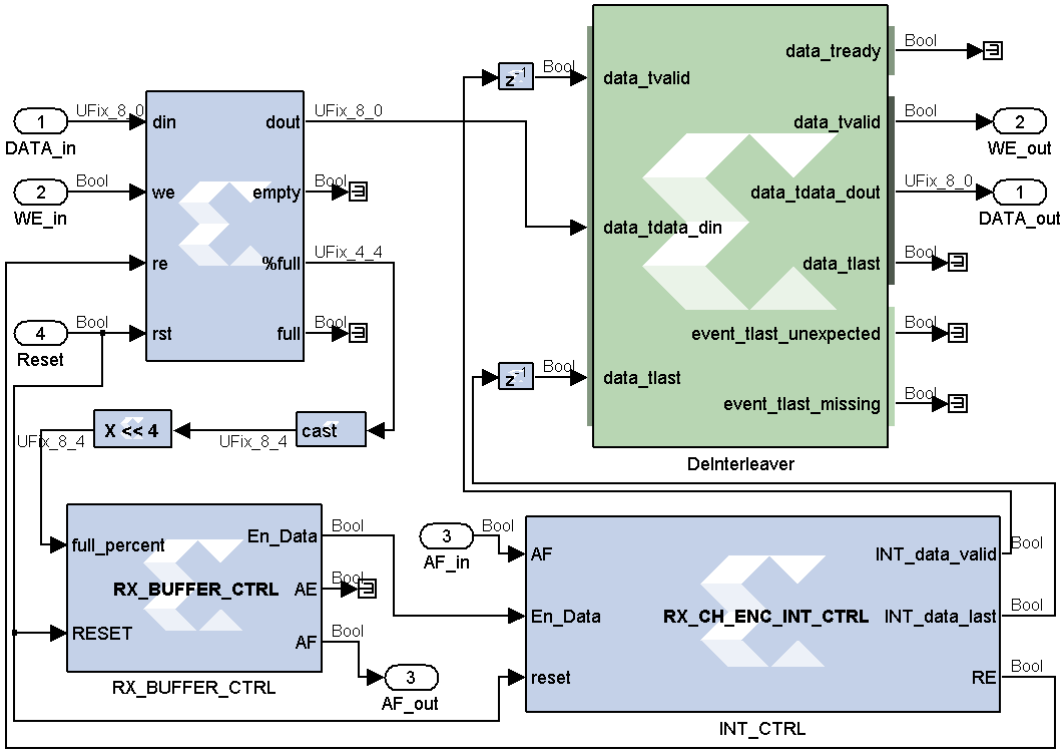

Figure 5.10: Asynchronous architecture - Block Deinterleaver

In Figure 5.10 is shown the Block Deinterleaver implementation. As the reader can see it has a similar structure when compared to the Interleaver. However, since this is the receiver side, the FIFO was moved from the output to the input. The size of the input buffer is 8KBytes and the RX_BUFFER_CTRL FSM defines the AF and AE as 75% and 25% respectively.

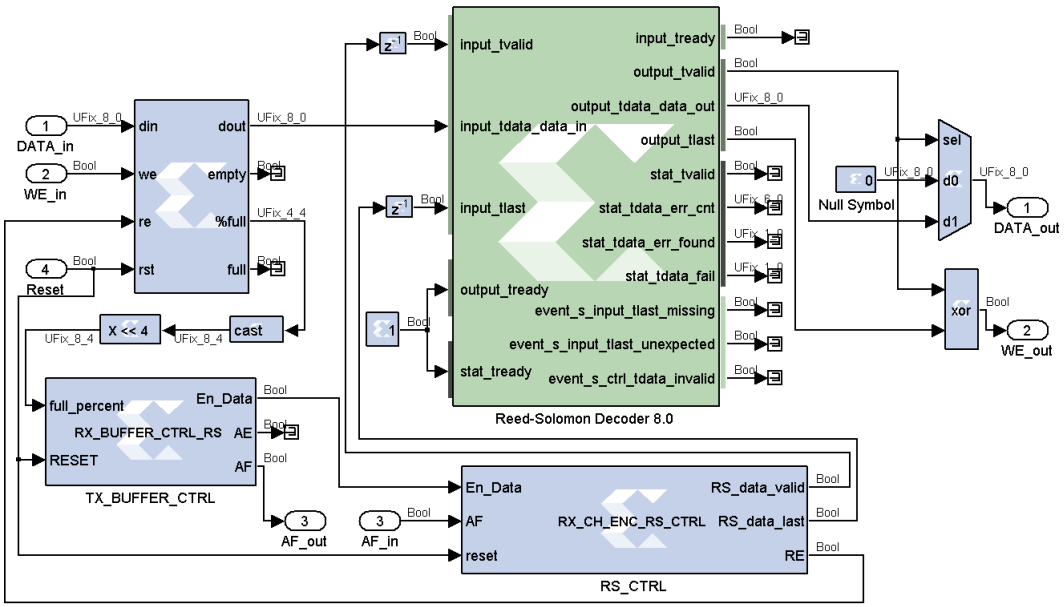

Figure 5.11: Asynchronous architecture - Reed-Solomon Decoder

The RS Decoder asynchronous block is shown in Figure 5.11. Three important remarks shall be made regarding this model. The first one is related to the FIFO storage capacity must take into account that are transmitted 2550 Bytes from Deinterleaver block at once. Thus, its value was setted as 8KBytes. Furthermore, the AF threshold was defined as 62,5% (3072 free Bytes) to accommodate a whole block Interleaver matrix. The second observation, already referred in the asynchronous RS Encoder, is that the decoder has a processing delay of 1074 symbol cycles. In other words, after the read of a complete codeword, the decoder needs to wait the processing delay time before start processing a new codeword. This is considered in the RX_CH_ENC_RS_CTRL FSM. Last but not the least, the junk symbol inserted in the encoder must now be discarded. The solution is to control the WE output signal, forcing it to zero when the last symbol is present in the RS decoder output.

After the RS error correction a CRC check will be performed. The CRC-32 Decoder, shown in Figure 5.12, is identical to the previous presented synchronous model with the respective changes in order to work in Bytes. The FIFO size at the input is 512 Bytes. The AF and AE thresholds was defined as 75% and 25% respectively. This block outputs the decoded data Bytes as well as the CRC_pass_fail and CRC_valid signals.

The FER is calculated in CRC Analysis block, shown in Figure 5.13. This block makes use of the error_rate_calc that was previously implemented. The outputs, number of received frames and number of frames containing errors are downsampled and stored in a "To FIFO" structure. The main goal is to generate the hardware co-simulation model for fast simulation results.

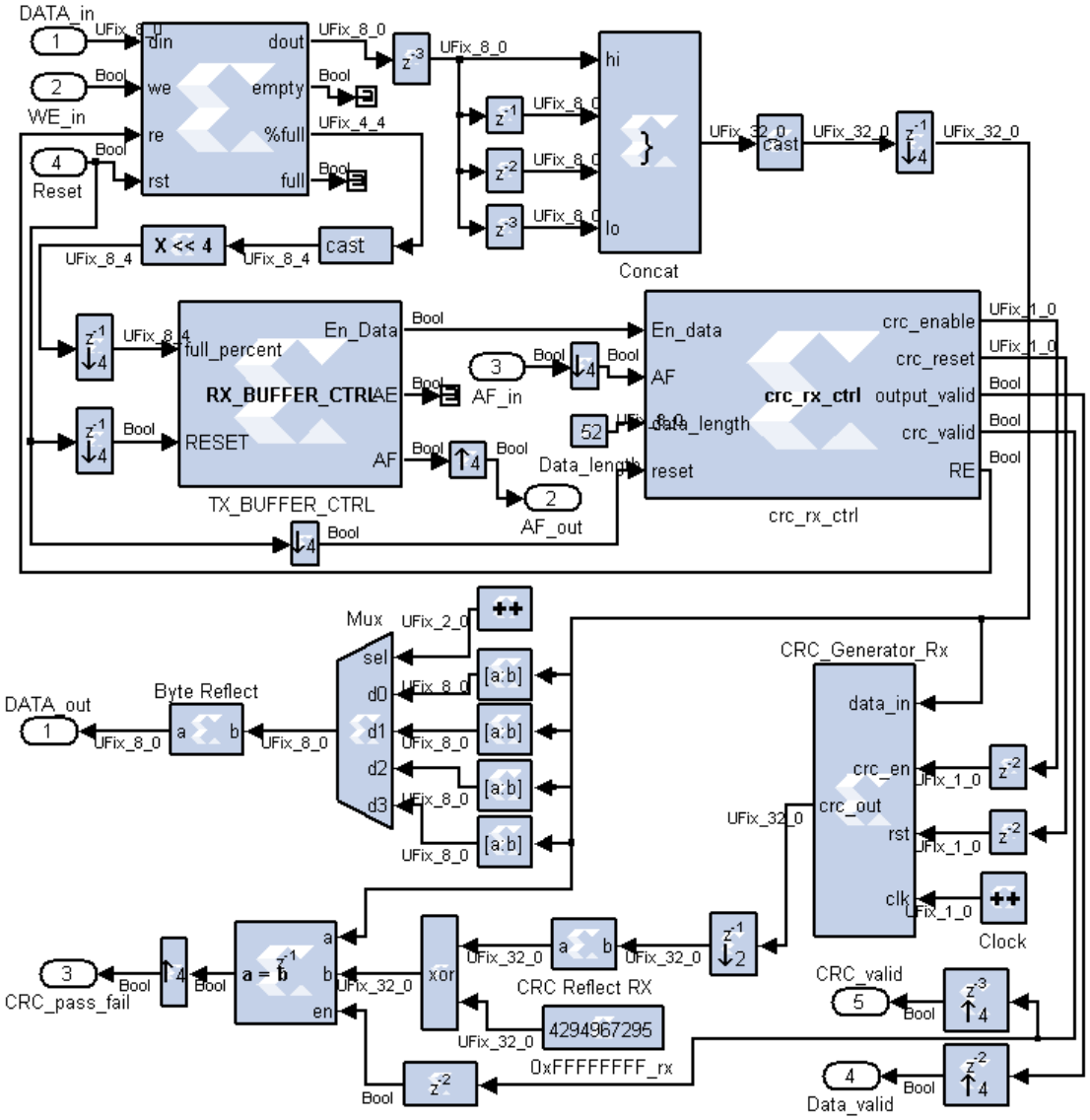

Figure 5.12: Asynchronous architecture - CRC-32 decoder

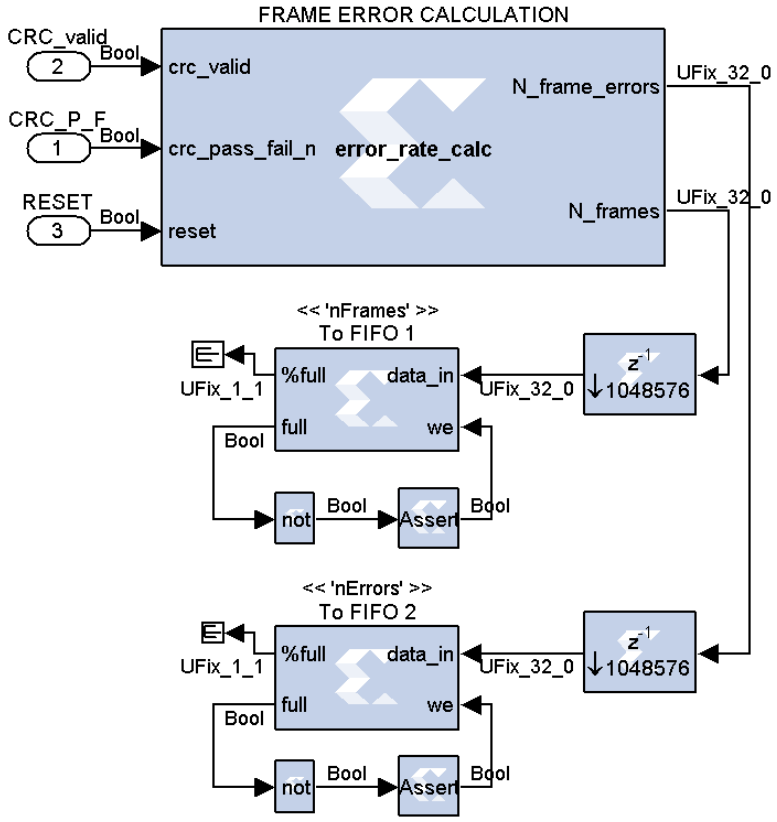

Figure 5.13: Asynchronous architecture - CRC result analysis

5.2 FPGA Utilization

One of the most important aspects in FPGA design is the number of used FPGA resources. In this particular work, since the proposed implementation aims to integrate in a larger system, a lower device utilization is even more important in order to spare resources for the other blocks of the project.

According to Figure 2.11, the system will be implemented in two separate FPGA's. Thus, the resources utilization of Channel Encoder and Channel Decoder shall be obtained separately.

Therefore, two models were created: one includes only the Channel Encoder and the other one the Channel Decoder. The device utilization results were obtained from the model compilation report. The Table 5.1 summarizes the FPGA resources used by the Channel Encoder.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	814	54576	1%
Number of Slice LUTs	981	27288	3%
Number used as Logic	951	27288	3%
Number used as Memory	30	6408	0%
Number used as SRL	30		
Slice Logic Distribution			
Number of LUT Flip Flop pairs used	1176		
Number with an unused Flip Flop	362	1176	30%
Number with an unused LUT	195	1176	16%
Number of fully used LUT-FF pairs	619	1176	52%
Number of unique control sets	38		
IO Utilization			
Number of IOs	3		
Number of bonded IOBs	0	296	0%
Specific Feature Utilization			
Number of Block RAM/FIFO	7	116	6%
Number using Block RAM only	7		
Number of BUFG/BUFGCTRL/BUFHCEs	1	16	6%

Table 5.1: Device Utilization Summary - Channel Encoder

As the reader can observe, the resources utilization percentage is very low, in the order of 3%. The FIFO usage is the higher value, about 6%. The Table 5.2 shows the device utilization of the Channel Decoder.

The results shows that the decoder uses more resources than the encoder. However, it only uses 3% of the total number of slice registers and 8% of slice LUTs (look-up tables).

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2011	54576	3%
Number of Slice LUTs	2267	27288	8%
Number used as Logic	2173	27288	7%
Number used as Memory	94	6408	1%
Number used as RAM	40		
Number used as SRL	54		
Slice Logic Distribution			
Number of LUT Flip Flop pairs used	2803		
Number with an unused Flip Flop	792	2803	28%
Number with an unused LUT	536	2803	19%
Number of fully used LUT-FF pairs	1475	2803	52%
Number of unique control sets	62		
IO Utilization			
Number of IOs	4		
Number of bonded IOBs	0	296	0%
Specific Feature Utilization			
Number of Block RAM/FIFO	13	116	11%
Number using Block RAM only	13		

Table 5.2: Device Utilization Summary - Channel Decoder

5.3 System Performance

In this section will be presented the asynchronous channel encoder model performance. It is important to note that due to the system complexity, a simulation in the Simulink environment would take several hours or even days to perform a single simulation. Therefore, the results in this section were obtained by using hardware co-simulation.

The first test performed in the system was its proper functioning. In this test, in order to maximize the transmission rate, the block Error_Generator was removed. Furthermore, the block Sync_output had some modification, shown in Figure 5.14, in order to be able to read up to 1 Byte per clock cycle.

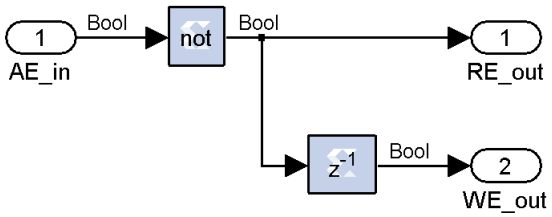

Figure 5.14: Asynchronous architecture - High data rate Sync Output block

The hardware co-simulation shows that all the 10^9 received frames were not affected by errors.

At this point it is interesting to measure what is the maximum throughput of the proposed

system. In order to obtain the system transmission rate an additional block was added to the System Generator model. This block, in Figure 5.15, is a simple accumulator of a sampled constant. It counts the number of clock cycles and stores them into a shared FIFO. The throughput can be calculated using the Equation 5.1, where $Frame\_size$ comes in Bytes and T_{clk} is the FPGA clock period in seconds.

$$R = \frac{N_{Frames} \times Payload\_size \times 8}{N_{Clocks} \times T_{clk}} \ (bit/s) \tag{5.1}$$

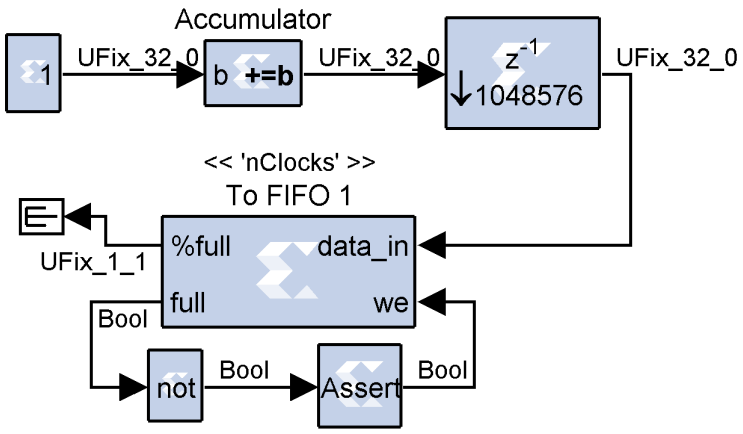

Figure 5.15: Clock Counter Block

The results from hardware co-simulation are shown in Table 5.3.

N_{Frames}	1752707
N_{Clocks}	2147483648

Table 5.3: System data rate results

From the Equation 5.1 and the values from Table 5.3, the system is capable to transmit up to 65,3 Mbit/s of data. The expected value is calculated from the maximum bit rate referred in Section 4.2.1 (95 Mbit/s) imposed by the RS decoder processing delay multiplied by the system efficiency from Equation 4.8 (78.43%), thus, 74.5 Mbit/s. The most probable cause for the lower measured value are the FIFO sizes of the asynchronous architecture as well as the AF and AE thresholds of buffer control FSMs. By using FIFOs with more storage capacity and redefine the FIFO thresholds, the throughput should get closer to the expected value. However, the actual VLCLighting results presents a transmission rate of 24 Mbit/s. Furthermore if we consider a system efficiency of 78.43%, the maximum throughput is 18,83 Mbit/s. Thus, the proposed Asynchronous Channel Encoder fulfil the required throughput of the VLCLighting project. Note that the present results were obtained with a clock frequency of 50 Mhz.

The system performance analysis would not be complete without the simulation of FER parameter. The results, shown in Figure 5.16, were obtained using the model of Figure 5.1, varying the Error Probability value. The estimated FER is obtained from Equations 4.6

and 4.5 with $N_{BLOCKS} = 1$. As the reader can see, the measured FER follows the theoretical value.

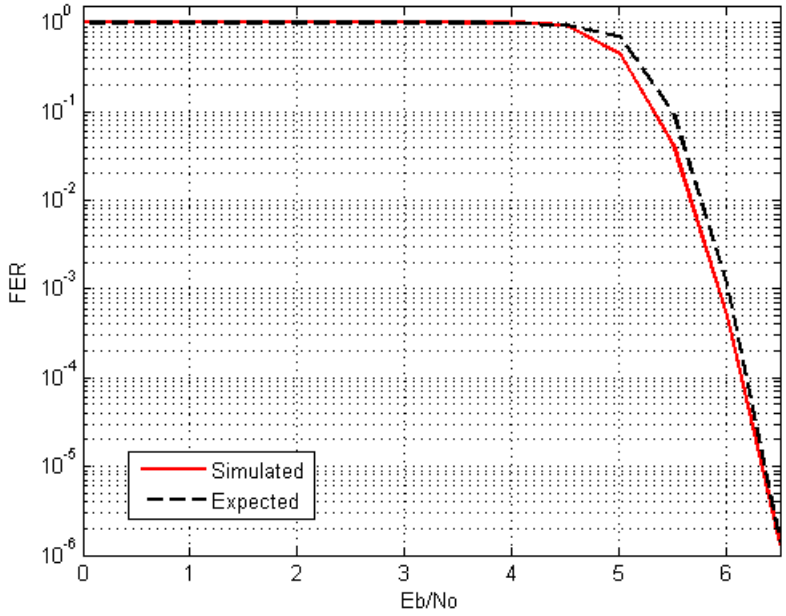

Figure 5.16: Frame Error Rate system performance

Chapter 6

Conclusions

6.1 Conclusions

In this work were discussed several FEC techniques. Later, it was made a study of the most appropriate combination of these techniques to implement, in hardware, a Channel Encoder comprising a CRC-32 computation block, a RS(213,255) code and a Block Interleaver.

The results shows that the usage of FEC techniques should always be considered in communication systems, particularly in broadcast systems. In these systems, is not possible to the emitter to guarantee that data was received without errors. Therefore, some preventive measures such as FEC schemes shall be taken in order to assure the proper system functionality.

The ECC are extremely important in order to recover corrupted data at the receiver side. The two studied ECC, Reed-Solomon and Convolutional codes, presented different characteristics. In one hand, Convolutional codes showed a great performance over random errors at low E_b/N_0 values. On the other hand, the RS codes perform better in the presence of burst errors since it corrects symbols instead of isolated bits. In particular, it was concluded that the RS(213,255) code performs better than a Convolutional code with K=7 and R=1/2 for higher values of E_b/N_0. Furthermore, this RS code has more efficiency than the referred Convolutional code.

In scenarios where the channel presents burst errors, such as in the case of the VLC, the usage of an interleaver scheme with the studied ECC usually improves the system performance. At the end of the decoding process, all errors may not be completely removed. The CRC-32 algorithm shows a great performance in the data validation.

The FPGA design was one of the most important aspects in this work. It proved that, despite all the additional work in the implementation process which requires to take care of the smallest details, it is a powerful tool to implement systems that require high-speed data processing. Furthermore, the hardware co-simulation helped in the simulation process, since it allows to obtain result faster when compared to the System Generator environment.

Another crucial implementation detail was the adoption the of Asynchronous Architecture. Its usage in the final implementation turn the interaction between blocks (CRC-32, RS and Interleaving) easier. It worked as expected: each block works independently of the adjacent blocks.

Concerning to the final Channel Encoder implementation performance, the FPGA resources utilization presents a low value. Therefore, the proposed Channel Encoder can be

inserted in a larger system without change significantly the FPGA utilization. Further results shows that the FER is according to the theoretical value.

6.2 Future Work

Through this work were considered several approaches in the Channel Encoder design. However, other aspects shall be further addressed in the future. Some of the most relevant future topics are listed below:

- Integrate the proposed Channel Encoder in the VLCLighting project.
- Real tests to evaluate system performance in a real scenario. Based on the results, implement changes in the Channel Encoder.
- Consider a RS code with less parity symbols. It should increase the payload size and at the same time it decreases the RS decoder processing delay value, which is the most limiting factor of the proposed system throughput.
- Make adjustments in the FIFOs size and thresholds (AF and AE) in order to allow the system to operate at the maximum speed without stop the processing units due to the threshold values.
- Additional studies for other ECC such as RS shortened codes, Concatenated codes, Turbo codes and Low-Density Parity-Check codes.

Bibliography

[1] G. Cossu, a. M. Khalid, P. Choudhury, R. Corsini, and E. Ciaramella, "3.4 Gbit/s visible optical wireless transmission based on RGB LED.," *Optics express*, vol. 20, pp. B501–6, Dec. 2012.

[2] C. Ribeiro, M. Figueiredo, L. N. Alves, L. Duarte, L. Rodrigues, and P. Rodrigues, "VLCLighting - A Collaborative Research Project on Visible Light Communication," 2015.

[3] "Visible Spectrum." http://chemistry.tutorcircle.com/inorganic-chemistry/visible-spectrum.html.

[4] Z. Ghassemlooy, W. Popoola, and S. Rajbhandari, *Optical wireless communications: system and channel modelling with Matlab*. 2012.

[5] D. Shin, D. K. Jung, Y. J. Oh, T. Bae, H.-c. Kwon, J. Son, and U. Kingdom, "Visible Light Communication : Tutorial," *Area*, no. March, 2008.

[6] PureVLC, "Differences Between Radio & Visible Light Communications." 2012.

[7] A. G. Bell, "Photophone." http://www.bluehaze.com.au/modlight/ModLightBiblio_-files/PhotoPFolk.gif.

[8] J. M. Kahn and J. R. Barry, "Wireless Infrared Communications," vol. 9219, no. 97, 1997.

[9] N. Zheludev, "Commentary. The life and times of the LED - a 100-year history," vol. 1, no. April, pp. 189–192, 2007.

[10] W. of LEDs, "What is the History of the LEDs - See a timeline of Milestones in LED Technology." http://www.worldofleds.co.uk/history-of-leds.htm.

[11] M. Craford, "LEDs challenge the incandescents," *IEEE Circuits and Devices Magazine*, vol. 8, 1992.

[12] I. Akasaki, H. Amano, A. Nobel, and W. Akasaki, "Blue LEDs Filling the world with new light,"

[13] R. Haitz, "Haitz' Law." http://www.parz.cn/uploaded_files/1232241746.jpg.

[14] R. Haitz and J. Y. Tsao, "Solid-state lighting: 'The case' 10 years after and future prospects," *Physica Status Solidi (A) Applications and Materials Science*, vol. 208, no. 1, pp. 17–29, 2011.

[15] S. Arnon, J. R. Barry, G. K. Karagiannidis, R. Schober, and M. Uysal, *Advanced Optical Wireless Communication Systems*. 2006.

[16] J.-p. Javaudin, "Home Gigabit Access," pp. 1–7, 2008.

[17] Fraunhofer HHI, "Visible Light Communication System up to 500 Mbit/s." http://www.hhi.fraunhofer.de/departments/photonic-networks-and-systems/products-and-services/vlc-system-500-mbits.html.

[18] PureLiFi, "pureLiFi,"

[19] M. Wright, "Philips Lighting deploys LED-based indoor positioning in Carrefour hypermarket," *LEDs Magazine*, 2015.

[20] S. Wu, H. Wang, and C.-h. Youn, "Visible light communications for 5G wireless networking systems: from fixed to mobile communications," *Network, IEEE*, no. December, pp. 41–45, 2014.

[21] G. Editor, R. L. Aguiar, I. D. Telecomunicações, R. L. Aguiar, J. S. Bedo, and B. Evans, "White Paper for Research Beyond 5G," no. October, pp. 1–43, 2015.

[22] Z. Wu, "Free Space Optical Networking with Visible Light: A Multi-hop Multi-access Solution," 2012.

[23] J. K. Kwon, "Inverse Source Coding for Dimming in Visible Light Communications Using NRZ-OOK on Reliable Links," *IEEE Photonics Technology Letters*, vol. 22, pp. 1455–1457, Oct. 2010.

[24] N. Saha and R. Mondal, "Mitigation of interference using OFDM in visible light communication," *ICT Convergence (ICTC), ...*, pp. 159–162, 2012.

[25] G. V. Meerbergen, M. Moonen, and H. D. Man, "Combining Reed-Solomon Codes and OFDM for Impulse Noise Mitigation: RS-OFDM," *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 4, pp. 657–660, 2006.

[26] H. Elgala, R. Mesleh, H. Haas, and B. Pricope, "OFDM Visible Light Wireless Communication Based on White LEDs," *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring*, pp. 2185–2189, Apr. 2007.

[27] I. Stefan and H. Elgala, "Optical Wireless OFDM System on FPGA: Study of LED Nonlinearity Effects," in *2011 IEEE 73rd Vehicular Technology Conference (VTC Spring)*, pp. 1–5, 2011.

[28] S. Dimitrov, S. Sinanovic, and H. Haas, "Clipping Noise in OFDM-Based Optical Wireless Communication Systems," *IEEE Transactions on Communications*, vol. 60, pp. 1072–1081, Apr. 2012.

[29] D. Tsonev, S. Sinanovic, and H. Haas, "Novel Unipolar Orthogonal Frequency Division Multiplexing (U-OFDM) for Optical Wireless," *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*, pp. 1–5, May 2012.

[30] M. Mehnert, D. V. Droste, and D. Schiel, "VHDL Implementation of a (255,191) Reed Solomon Coder for DVB-H," *2006 IEEE International Symposium on Consumer Electronics*, no. 2, 2006.

[31] Y. L. Y. Liu, Y. G. Y. Guan, J. Z. J. Zhang, G. W. G. Wang, and Y. Z. Y. Zhang, "Reed-Solomon Codes for Satellite Communications," *2009 IITA International Conference on Control, Automation and Systems Engineering (case 2009)*, pp. 246–249, 2009.

[32] Mit.edu, "Convolutional Coding." http://web.mit.edu/6.02/www/s2010/handouts/lectures/L8-notes.pdf, 2010.

[33] Mit.edu, "Viterbi Decoding of Convolutional Codes." http://web.mit.edu/6.02/www/s2010/handouts/lectures/L9-notes.pdf, 2010.

[34] F. Jiang, E. Psota, and L. C. Pérez, "Decoding turbo codes based on their parity-check matrices," *Proceedings of the Annual Southeastern Symposium on System Theory*, pp. 221–224, 2007.

[35] C. Berrou, R. Pyndiah, P. Adde, C. Douillard, and R. L. Bidan, "An overview of turbo codes and their applications," *The European Conference on Wireless Technology, 2005.*, 2005.

[36] B.-S. S. B.-S. Shim, S.-J. C. S.-J. Choi, H.-K. P. H.-K. Park, S.-Y. K. S.-Y. Kim, and Y.-C. R. Y.-C. Ra, "A Study on Performance Evaluation of the Asymmetric Turbo Codes," *2008 International Conference on Convergence and Hybrid Information Technology*, pp. 667–671, 2008.

[37] J. J. J. Jiang and K. Narayanan, "Iterative soft decoding of Reed-Solomon codes," *IEEE Communications Letters*, vol. 8, no. 4, 2004.

[38] Y. Q. Shi, X. M. Zhang, Z. C. Ni, and N. Ansari, "Interleaving for combating bursts of errors," *IEEE Circuits and Systems Magazine*, vol. 4, pp. 29–42, 2004.

[39] V. Tomashevich, "Convolutional Codes," pp. 2–7, 2005.

[40] B. Lee and S. Choi, *Broadband wireless access and local networks: mobile WiMAX and WiFi*. 2008.

[41] R. D. Roberts, "IEEE 802 . 15 . 7 Visible Light Communication : Modulation Schemes and Dimming Support," *IEEE communications magazine*, no. March, pp. 72–82, 2012.

[42] "FPGA structure." http://www.xilinx.com/images/fpga-block-structure.gif.

[43] G. Hasslinger and O. Hohlfeld, "The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet," 2008.

[44] B. Sklar, *DIGITAL Fundamentals and Applications*. 2013.

[45] "Reed Solmon Decoder." http://cdstahl.org/wp-content/uploads/2010/08/rsd_-block.png.

[46] S. b. Wicker, "Error control systems for digital communication and storage," 1995.

[47] J. G. Proakis, *Digital Communications*. 3rd ed., 1989.

[48] "Convolutional Codes." http://www.tdm.uni-oldenburg.de/2004/Material/faltung.htm.

[49] "Convolutional Interleaver." https://awrcorp.com/download/faq/english/docs/VSS_-System_Blocks/images/cnv_dlvr_fig1.png.

[50] Xilinx, "Xilinx LogiCORE IP Reed-Solomon Decoder v8.0, Data Sheet," pp. 1–32, 2011.

[51] Xilinx, "LogiCORE IP Convolutional Encoder v8.0," pp. 0–22, 2012.

[52] Xilinx, "LogiCORE IP Viterbi Decoder v7.0, DataSheet," pp. 1–34, 2011.

[53] Xilinx, "LogiCORE IP Interleaver/ De-Interleaver v7.1 Product Guide," pp. 1–189, 2013.

[54] E. Stavinov, "A Practical Parallel CRC Generation Method," 2010.

[55] Outputlogic.com, "outputlogic.com - CRC generator." http://outputlogic.com/?page_-id=321.

[56] P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo- Random Sequence Generators," *Xilinx*, vol. 1996, pp. 1–6, 1996.

[57] Tektronix, "A Guide to MPEG Fundamentals and Protocol Analysis,"

[58] C. Ribeiro, M. Figueiredo, and L. N. Alves, "A Real-Time Platform for Collaborative Research on Visible Light Communication," 2015.

Appendix A

CRC-32 model

Parallel CRC generator VHDL code

```
1  ------------------------------------------------------------------------------
2  -- Copyright (C) 2009 OutputLogic.com
3  -- This source file may be used and distributed without restriction
4  -- provided that this copyright statement is not removed from the file
5  -- and that any derivative work contains the original copyright notice
6  -- and the associated disclaimer.
7  --
8  -- THIS SOURCE FILE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS
9  -- OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
10 -- WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
11 ------------------------------------------------------------------------------
12 -- CRC module for data(31:0)
13 --   ...
       lfsr(31:0)=1+x^1+x^2+x^4+x^5+x^7+x^8+x^10+x^11+x^12+x^16+x^22+x^23+x^26+x^32;
14 ------------------------------------------------------------------------------
15 library ieee;
16 use ieee.std_logic_1164.all;
17
18 entity crc is
19   port ( data_in : in std_logic_vector (31 downto 0);
20     crc_en , rst, clk : in std_logic;
21     crc_out : out std_logic_vector (31 downto 0));
22 end crc;
23
24 architecture imp_crc of crc is
25   signal lfsr_q: std_logic_vector (31 downto 0);
26   signal lfsr_c: std_logic_vector (31 downto 0);
27 begin
28     crc_out <= lfsr_q;
29
30     lfsr_c(0) <= lfsr_q(0) xor lfsr_q(6) xor lfsr_q(9) xor lfsr_q(10) xor ...
           lfsr_q(12) xor lfsr_q(16) xor lfsr_q(24) xor lfsr_q(25) xor lfsr_q(26) ...
           xor lfsr_q(28) xor lfsr_q(29) xor lfsr_q(30) xor lfsr_q(31) xor ...
           data_in(0) xor data_in(6) xor data_in(9) xor data_in(10) xor ...
           data_in(12) xor data_in(16) xor data_in(24) xor data_in(25) xor ...
           data_in(26) xor data_in(28) xor data_in(29) xor data_in(30) xor ...
           data_in(31);
```

```
31      lfsr_c(1) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(6) xor lfsr_q(7) xor ...
            lfsr_q(9) xor lfsr_q(11) xor lfsr_q(12) xor lfsr_q(13) xor lfsr_q(16) ...
            xor lfsr_q(17) xor lfsr_q(24) xor lfsr_q(27) xor lfsr_q(28) xor ...
            data_in(0) xor data_in(1) xor data_in(6) xor data_in(7) xor data_in(9) ...
            xor data_in(11) xor data_in(12) xor data_in(13) xor data_in(16) xor ...
            data_in(17) xor data_in(24) xor data_in(27) xor data_in(28);
32      lfsr_c(2) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(2) xor lfsr_q(6) xor ...
            lfsr_q(7) xor lfsr_q(8) xor lfsr_q(9) xor lfsr_q(13) xor lfsr_q(14) ...
            xor lfsr_q(16) xor lfsr_q(17) xor lfsr_q(18) xor lfsr_q(24) xor ...
            lfsr_q(26) xor lfsr_q(30) xor lfsr_q(31) xor data_in(0) xor data_in(1) ...
            xor data_in(2) xor data_in(6) xor data_in(7) xor data_in(8) xor ...
            data_in(9) xor data_in(13) xor data_in(14) xor data_in(16) xor ...
            data_in(17) xor data_in(18) xor data_in(24) xor data_in(26) xor ...
            data_in(30) xor data_in(31);
33      lfsr_c(3) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(7) xor ...
            lfsr_q(8) xor lfsr_q(9) xor lfsr_q(10) xor lfsr_q(14) xor lfsr_q(15) ...
            xor lfsr_q(17) xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(25) xor ...
            lfsr_q(27) xor lfsr_q(31) xor data_in(1) xor data_in(2) xor data_in(3) ...
            xor data_in(7) xor data_in(8) xor data_in(9) xor data_in(10) xor ...
            data_in(14) xor data_in(15) xor data_in(17) xor data_in(18) xor ...
            data_in(19) xor data_in(25) xor data_in(27) xor data_in(31);
34      lfsr_c(4) <= lfsr_q(0) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(4) xor ...
            lfsr_q(6) xor lfsr_q(8) xor lfsr_q(11) xor lfsr_q(12) xor lfsr_q(15) ...
            xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(20) xor lfsr_q(24) xor ...
            lfsr_q(25) xor lfsr_q(29) xor lfsr_q(30) xor lfsr_q(31) xor data_in(0) ...
            xor data_in(2) xor data_in(3) xor data_in(4) xor data_in(6) xor ...
            data_in(8) xor data_in(11) xor data_in(12) xor data_in(15) xor ...
            data_in(18) xor data_in(19) xor data_in(20) xor data_in(24) xor ...
            data_in(25) xor data_in(29) xor data_in(30) xor data_in(31);
35      lfsr_c(5) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(3) xor lfsr_q(4) xor ...
            lfsr_q(5) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(10) xor lfsr_q(13) ...
            xor lfsr_q(19) xor lfsr_q(20) xor lfsr_q(21) xor lfsr_q(24) xor ...
            lfsr_q(28) xor lfsr_q(29) xor data_in(0) xor data_in(1) xor data_in(3) ...
            xor data_in(4) xor data_in(5) xor data_in(6) xor data_in(7) xor ...
            data_in(10) xor data_in(13) xor data_in(19) xor data_in(20) xor ...
            data_in(21) xor data_in(24) xor data_in(28) xor data_in(29);
36      lfsr_c(6) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(4) xor lfsr_q(5) xor ...
            lfsr_q(6) xor lfsr_q(7) xor lfsr_q(8) xor lfsr_q(11) xor lfsr_q(14) ...
            xor lfsr_q(20) xor lfsr_q(21) xor lfsr_q(22) xor lfsr_q(25) xor ...
            lfsr_q(29) xor lfsr_q(30) xor data_in(1) xor data_in(2) xor data_in(4) ...
            xor data_in(5) xor data_in(6) xor data_in(7) xor data_in(8) xor ...
            data_in(11) xor data_in(14) xor data_in(20) xor data_in(21) xor ...
            data_in(22) xor data_in(25) xor data_in(29) xor data_in(30);
37      lfsr_c(7) <= lfsr_q(0) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(5) xor ...
            lfsr_q(7) xor lfsr_q(8) xor lfsr_q(10) xor lfsr_q(15) xor lfsr_q(16) ...
            xor lfsr_q(21) xor lfsr_q(22) xor lfsr_q(23) xor lfsr_q(24) xor ...
            lfsr_q(25) xor lfsr_q(28) xor lfsr_q(29) xor data_in(0) xor data_in(2) ...
            xor data_in(3) xor data_in(5) xor data_in(7) xor data_in(8) xor ...
            data_in(10) xor data_in(15) xor data_in(16) xor data_in(21) xor ...
            data_in(22) xor data_in(23) xor data_in(24) xor data_in(25) xor ...
            data_in(28) xor data_in(29);
38      lfsr_c(8) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(3) xor lfsr_q(4) xor ...
            lfsr_q(8) xor lfsr_q(10) xor lfsr_q(11) xor lfsr_q(12) xor lfsr_q(17) ...
            xor lfsr_q(22) xor lfsr_q(23) xor lfsr_q(28) xor lfsr_q(31) xor ...
            data_in(0) xor data_in(1) xor data_in(3) xor data_in(4) xor data_in(8) ...
            xor data_in(10) xor data_in(11) xor data_in(12) xor data_in(17) xor ...
            data_in(22) xor data_in(23) xor data_in(28) xor data_in(31);
```

```
39      lfsr_c(9) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(4) xor lfsr_q(5) xor ...
            lfsr_q(9) xor lfsr_q(11) xor lfsr_q(12) xor lfsr_q(13) xor lfsr_q(18) ...
            xor lfsr_q(23) xor lfsr_q(24) xor lfsr_q(29) xor data_in(1) xor ...
            data_in(2) xor data_in(4) xor data_in(5) xor data_in(9) xor ...
            data_in(11) xor data_in(12) xor data_in(13) xor data_in(18) xor ...
            data_in(23) xor data_in(24) xor data_in(29);
40      lfsr_c(10) <= lfsr_q(0) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(5) xor ...
            lfsr_q(9) xor lfsr_q(13) xor lfsr_q(14) xor lfsr_q(16) xor lfsr_q(19) ...
            xor lfsr_q(26) xor lfsr_q(28) xor lfsr_q(29) xor lfsr_q(31) xor ...
            data_in(0) xor data_in(2) xor data_in(3) xor data_in(5) xor data_in(9) ...
            xor data_in(13) xor data_in(14) xor data_in(16) xor data_in(19) xor ...
            data_in(26) xor data_in(28) xor data_in(29) xor data_in(31);
41      lfsr_c(11) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(3) xor lfsr_q(4) xor ...
            lfsr_q(9) xor lfsr_q(12) xor lfsr_q(14) xor lfsr_q(15) xor lfsr_q(16) ...
            xor lfsr_q(17) xor lfsr_q(20) xor lfsr_q(24) xor lfsr_q(25) xor ...
            lfsr_q(26) xor lfsr_q(27) xor lfsr_q(28) xor lfsr_q(31) xor data_in(0) ...
            xor data_in(1) xor data_in(3) xor data_in(4) xor data_in(9) xor ...
            data_in(12) xor data_in(14) xor data_in(15) xor data_in(16) xor ...
            data_in(17) xor data_in(20) xor data_in(24) xor data_in(25) xor ...
            data_in(26) xor data_in(27) xor data_in(28) xor data_in(31);
42      lfsr_c(12) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(2) xor lfsr_q(4) xor ...
            lfsr_q(5) xor lfsr_q(6) xor lfsr_q(9) xor lfsr_q(12) xor lfsr_q(13) ...
            xor lfsr_q(15) xor lfsr_q(17) xor lfsr_q(18) xor lfsr_q(21) xor ...
            lfsr_q(24) xor lfsr_q(27) xor lfsr_q(30) xor lfsr_q(31) xor data_in(0) ...
            xor data_in(1) xor data_in(2) xor data_in(4) xor data_in(5) xor ...
            data_in(6) xor data_in(9) xor data_in(12) xor data_in(13) xor ...
            data_in(15) xor data_in(17) xor data_in(18) xor data_in(21) xor ...
            data_in(24) xor data_in(27) xor data_in(30) xor data_in(31);
43      lfsr_c(13) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(3) xor lfsr_q(5) xor ...
            lfsr_q(6) xor lfsr_q(7) xor lfsr_q(10) xor lfsr_q(13) xor lfsr_q(14) ...
            xor lfsr_q(16) xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(22) xor ...
            lfsr_q(25) xor lfsr_q(28) xor lfsr_q(31) xor data_in(1) xor data_in(2) ...
            xor data_in(3) xor data_in(5) xor data_in(6) xor data_in(7) xor ...
            data_in(10) xor data_in(13) xor data_in(14) xor data_in(16) xor ...
            data_in(18) xor data_in(19) xor data_in(22) xor data_in(25) xor ...
            data_in(28) xor data_in(31);
44      lfsr_c(14) <= lfsr_q(2) xor lfsr_q(3) xor lfsr_q(4) xor lfsr_q(6) xor ...
            lfsr_q(7) xor lfsr_q(8) xor lfsr_q(11) xor lfsr_q(14) xor lfsr_q(15) ...
            xor lfsr_q(17) xor lfsr_q(19) xor lfsr_q(20) xor lfsr_q(23) xor ...
            lfsr_q(26) xor lfsr_q(29) xor data_in(2) xor data_in(3) xor data_in(4) ...
            xor data_in(6) xor data_in(7) xor data_in(8) xor data_in(11) xor ...
            data_in(14) xor data_in(15) xor data_in(17) xor data_in(19) xor ...
            data_in(20) xor data_in(23) xor data_in(26) xor data_in(29);
45      lfsr_c(15) <= lfsr_q(3) xor lfsr_q(4) xor lfsr_q(5) xor lfsr_q(7) xor ...
            lfsr_q(8) xor lfsr_q(9) xor lfsr_q(12) xor lfsr_q(15) xor lfsr_q(16) ...
            xor lfsr_q(18) xor lfsr_q(20) xor lfsr_q(21) xor lfsr_q(24) xor ...
            lfsr_q(27) xor lfsr_q(30) xor data_in(3) xor data_in(4) xor data_in(5) ...
            xor data_in(7) xor data_in(8) xor data_in(9) xor data_in(12) xor ...
            data_in(15) xor data_in(16) xor data_in(18) xor data_in(20) xor ...
            data_in(21) xor data_in(24) xor data_in(27) xor data_in(30);
46      lfsr_c(16) <= lfsr_q(0) xor lfsr_q(4) xor lfsr_q(5) xor lfsr_q(8) xor ...
            lfsr_q(12) xor lfsr_q(13) xor lfsr_q(17) xor lfsr_q(19) xor lfsr_q(21) ...
            xor lfsr_q(22) xor lfsr_q(24) xor lfsr_q(26) xor lfsr_q(29) xor ...
            lfsr_q(30) xor data_in(0) xor data_in(4) xor data_in(5) xor data_in(8) ...
            xor data_in(12) xor data_in(13) xor data_in(17) xor data_in(19) xor ...
            data_in(21) xor data_in(22) xor data_in(24) xor data_in(26) xor ...
            data_in(29) xor data_in(30);
```

```
47      lfsr_c(17) <= lfsr_q(1) xor lfsr_q(5) xor lfsr_q(6) xor lfsr_q(9) xor ...
            lfsr_q(13) xor lfsr_q(14) xor lfsr_q(18) xor lfsr_q(20) xor lfsr_q(22) ...
            xor lfsr_q(23) xor lfsr_q(25) xor lfsr_q(27) xor lfsr_q(30) xor ...
            lfsr_q(31) xor data_in(1) xor data_in(5) xor data_in(6) xor data_in(9) ...
            xor data_in(13) xor data_in(14) xor data_in(18) xor data_in(20) xor ...
            data_in(22) xor data_in(23) xor data_in(25) xor data_in(27) xor ...
            data_in(30) xor data_in(31);
48      lfsr_c(18) <= lfsr_q(2) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(10) xor ...
            lfsr_q(14) xor lfsr_q(15) xor lfsr_q(19) xor lfsr_q(21) xor lfsr_q(23) ...
            xor lfsr_q(24) xor lfsr_q(26) xor lfsr_q(28) xor lfsr_q(31) xor ...
            data_in(2) xor data_in(6) xor data_in(7) xor data_in(10) xor ...
            data_in(14) xor data_in(15) xor data_in(19) xor data_in(21) xor ...
            data_in(23) xor data_in(24) xor data_in(26) xor data_in(28) xor ...
            data_in(31);
49      lfsr_c(19) <= lfsr_q(3) xor lfsr_q(7) xor lfsr_q(8) xor lfsr_q(11) xor ...
            lfsr_q(15) xor lfsr_q(16) xor lfsr_q(20) xor lfsr_q(22) xor lfsr_q(24) ...
            xor lfsr_q(25) xor lfsr_q(27) xor lfsr_q(29) xor data_in(3) xor ...
            data_in(7) xor data_in(8) xor data_in(11) xor data_in(15) xor ...
            data_in(16) xor data_in(20) xor data_in(22) xor data_in(24) xor ...
            data_in(25) xor data_in(27) xor data_in(29);
50      lfsr_c(20) <= lfsr_q(4) xor lfsr_q(8) xor lfsr_q(9) xor lfsr_q(12) xor ...
            lfsr_q(16) xor lfsr_q(17) xor lfsr_q(21) xor lfsr_q(23) xor lfsr_q(25) ...
            xor lfsr_q(26) xor lfsr_q(28) xor lfsr_q(30) xor data_in(4) xor ...
            data_in(8) xor data_in(9) xor data_in(12) xor data_in(16) xor ...
            data_in(17) xor data_in(21) xor data_in(23) xor data_in(25) xor ...
            data_in(26) xor data_in(28) xor data_in(30);
51      lfsr_c(21) <= lfsr_q(5) xor lfsr_q(9) xor lfsr_q(10) xor lfsr_q(13) xor ...
            lfsr_q(17) xor lfsr_q(18) xor lfsr_q(22) xor lfsr_q(24) xor lfsr_q(26) ...
            xor lfsr_q(27) xor lfsr_q(29) xor lfsr_q(31) xor data_in(5) xor ...
            data_in(9) xor data_in(10) xor data_in(13) xor data_in(17) xor ...
            data_in(18) xor data_in(22) xor data_in(24) xor data_in(26) xor ...
            data_in(27) xor data_in(29) xor data_in(31);
52      lfsr_c(22) <= lfsr_q(0) xor lfsr_q(9) xor lfsr_q(11) xor lfsr_q(12) xor ...
            lfsr_q(14) xor lfsr_q(16) xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(23) ...
            xor lfsr_q(24) xor lfsr_q(26) xor lfsr_q(27) xor lfsr_q(29) xor ...
            lfsr_q(31) xor data_in(0) xor data_in(9) xor data_in(11) xor ...
            data_in(12) xor data_in(14) xor data_in(16) xor data_in(18) xor ...
            data_in(19) xor data_in(23) xor data_in(24) xor data_in(26) xor ...
            data_in(27) xor data_in(29) xor data_in(31);
53      lfsr_c(23) <= lfsr_q(0) xor lfsr_q(1) xor lfsr_q(6) xor lfsr_q(9) xor ...
            lfsr_q(13) xor lfsr_q(15) xor lfsr_q(16) xor lfsr_q(17) xor lfsr_q(19) ...
            xor lfsr_q(20) xor lfsr_q(26) xor lfsr_q(27) xor lfsr_q(29) xor ...
            lfsr_q(31) xor data_in(0) xor data_in(1) xor data_in(6) xor data_in(9) ...
            xor data_in(13) xor data_in(15) xor data_in(16) xor data_in(17) xor ...
            data_in(19) xor data_in(20) xor data_in(26) xor data_in(27) xor ...
            data_in(29) xor data_in(31);
54      lfsr_c(24) <= lfsr_q(1) xor lfsr_q(2) xor lfsr_q(7) xor lfsr_q(10) xor ...
            lfsr_q(14) xor lfsr_q(16) xor lfsr_q(17) xor lfsr_q(18) xor lfsr_q(20) ...
            xor lfsr_q(21) xor lfsr_q(27) xor lfsr_q(28) xor lfsr_q(30) xor ...
            data_in(1) xor data_in(2) xor data_in(7) xor data_in(10) xor ...
            data_in(14) xor data_in(16) xor data_in(17) xor data_in(18) xor ...
            data_in(20) xor data_in(21) xor data_in(27) xor data_in(28) xor ...
            data_in(30);
55      lfsr_c(25) <= lfsr_q(2) xor lfsr_q(3) xor lfsr_q(8) xor lfsr_q(11) xor ...
            lfsr_q(15) xor lfsr_q(17) xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(21) ...
            xor lfsr_q(22) xor lfsr_q(28) xor lfsr_q(29) xor lfsr_q(31) xor ...
            data_in(2) xor data_in(3) xor data_in(8) xor data_in(11) xor ...
```

```
          data_in(15) xor data_in(17) xor data_in(18) xor data_in(19) xor ...
          data_in(21) xor data_in(22) xor data_in(28) xor data_in(29) xor ...
          data_in(31);
56      lfsr_c(26) <= lfsr_q(0) xor lfsr_q(3) xor lfsr_q(4) xor lfsr_q(6) xor ...
          lfsr_q(10) xor lfsr_q(18) xor lfsr_q(19) xor lfsr_q(20) xor lfsr_q(22) ...
          xor lfsr_q(23) xor lfsr_q(24) xor lfsr_q(25) xor lfsr_q(26) xor ...
          lfsr_q(28) xor lfsr_q(31) xor data_in(0) xor data_in(3) xor data_in(4) ...
          xor data_in(6) xor data_in(10) xor data_in(18) xor data_in(19) xor ...
          data_in(20) xor data_in(22) xor data_in(23) xor data_in(24) xor ...
          data_in(25) xor data_in(26) xor data_in(28) xor data_in(31);
57      lfsr_c(27) <= lfsr_q(1) xor lfsr_q(4) xor lfsr_q(5) xor lfsr_q(7) xor ...
          lfsr_q(11) xor lfsr_q(19) xor lfsr_q(20) xor lfsr_q(21) xor lfsr_q(23) ...
          xor lfsr_q(24) xor lfsr_q(25) xor lfsr_q(26) xor lfsr_q(27) xor ...
          lfsr_q(29) xor data_in(1) xor data_in(4) xor data_in(5) xor data_in(7) ...
          xor data_in(11) xor data_in(19) xor data_in(20) xor data_in(21) xor ...
          data_in(23) xor data_in(24) xor data_in(25) xor data_in(26) xor ...
          data_in(27) xor data_in(29);
58      lfsr_c(28) <= lfsr_q(2) xor lfsr_q(5) xor lfsr_q(6) xor lfsr_q(8) xor ...
          lfsr_q(12) xor lfsr_q(20) xor lfsr_q(21) xor lfsr_q(22) xor lfsr_q(24) ...
          xor lfsr_q(25) xor lfsr_q(26) xor lfsr_q(27) xor lfsr_q(28) xor ...
          lfsr_q(30) xor data_in(2) xor data_in(5) xor data_in(6) xor data_in(8) ...
          xor data_in(12) xor data_in(20) xor data_in(21) xor data_in(22) xor ...
          data_in(24) xor data_in(25) xor data_in(26) xor data_in(27) xor ...
          data_in(28) xor data_in(30);
59      lfsr_c(29) <= lfsr_q(3) xor lfsr_q(6) xor lfsr_q(7) xor lfsr_q(9) xor ...
          lfsr_q(13) xor lfsr_q(21) xor lfsr_q(22) xor lfsr_q(23) xor lfsr_q(25) ...
          xor lfsr_q(26) xor lfsr_q(27) xor lfsr_q(28) xor lfsr_q(29) xor ...
          lfsr_q(31) xor data_in(3) xor data_in(6) xor data_in(7) xor data_in(9) ...
          xor data_in(13) xor data_in(21) xor data_in(22) xor data_in(23) xor ...
          data_in(25) xor data_in(26) xor data_in(27) xor data_in(28) xor ...
          data_in(29) xor data_in(31);
60      lfsr_c(30) <= lfsr_q(4) xor lfsr_q(7) xor lfsr_q(8) xor lfsr_q(10) xor ...
          lfsr_q(14) xor lfsr_q(22) xor lfsr_q(23) xor lfsr_q(24) xor lfsr_q(26) ...
          xor lfsr_q(27) xor lfsr_q(28) xor lfsr_q(29) xor lfsr_q(30) xor ...
          data_in(4) xor data_in(7) xor data_in(8) xor data_in(10) xor ...
          data_in(14) xor data_in(22) xor data_in(23) xor data_in(24) xor ...
          data_in(26) xor data_in(27) xor data_in(28) xor data_in(29) xor ...
          data_in(30);
61      lfsr_c(31) <= lfsr_q(5) xor lfsr_q(8) xor lfsr_q(9) xor lfsr_q(11) xor ...
          lfsr_q(15) xor lfsr_q(23) xor lfsr_q(24) xor lfsr_q(25) xor lfsr_q(27) ...
          xor lfsr_q(28) xor lfsr_q(29) xor lfsr_q(30) xor lfsr_q(31) xor ...
          data_in(5) xor data_in(8) xor data_in(9) xor data_in(11) xor ...
          data_in(15) xor data_in(23) xor data_in(24) xor data_in(25) xor ...
          data_in(27) xor data_in(28) xor data_in(29) xor data_in(30) xor ...
          data_in(31);
62
63
64      process (clk,rst) begin
65        if (rst = '1') then
66          lfsr_q <= b"11111111111111111111111111111111";
67        elsif (clk'EVENT and clk = '1') then
68          if (crc_en = '1') then
69            lfsr_q <= lfsr_c;
70          end if;
71        end if;
72      end process;
73  end architecture imp_crc;
```

MatLab code for CRC emitter state machine controller

```
%% Emitter CRC generator control state machine
function [ data_out_select, crc_enable, crc_reset, output_valid ] = ...
    crc_tx_ctrl( data_valid, data_length )

    RESET = 0;
    DATA_AQ = 1;
    CRC_WAIT = 2;
    WAIT = 3;

    DATA = 0;
    CRC = 1;

    persistent state, state = xl_state(RESET,{xlUnsigned, 2, 0});
    persistent data_out_select_internal, data_out_select_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent crc_enable_internal, crc_enable_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent data_counter, data_counter = xl_state(0,{xlUnsigned, 8, 0});
    persistent crc_reset_internal, crc_reset_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent output_valid_internal, output_valid_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});

        switch state
            case RESET
                data_out_select_internal = DATA;
                crc_enable_internal = 1;
                crc_reset_internal = 1;
                output_valid_internal = 0;

                data_counter = 1;
                if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},data_valid) == 1
                    state = DATA_AQ;
                end


            case DATA_AQ
                if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},data_valid) == 1
                    data_out_select_internal = DATA;
                    crc_enable_internal = 1;
                    crc_reset_internal = 0;
                    output_valid_internal = 1;

                    data_counter = xfix({xlUnsigned, 8, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);

                    if xfix({xlUnsigned, 8, 0, xlTruncate, ...
                        xlWrap},data_counter) ≥ xfix({xlUnsigned, 8, 0, ...
                        xlTruncate, xlWrap},data_length-1)
                        state = CRC_WAIT;
                    end
                end


```

```
47              case CRC_WAIT
48                  data_out_select_internal = CRC;
49                  crc_enable_internal = 1;
50                  crc_reset_internal = 0;
51                  output_valid_internal = 1;
52
53                  state = RESET;
54
55
56               case WAIT
57                  data_out_select_internal = CRC;
58                  crc_enable_internal = 0;
59                  crc_reset_internal = 1;
60                  output_valid_internal = 0;
61
62                  state = RESET;
63
64              otherwise
65                  state = RESET;
66          end
67
68      data_out_select = data_out_select_internal;
69      crc_enable = crc_enable_internal;
70      crc_reset = crc_reset_internal;
71      output_valid = output_valid_internal;
72  end
```

MatLab code for CRC receiver state machine controller

```
function [crc_enable, crc_reset, output_valid, crc_valid] = crc_rx_ctrl( ...
    data_valid, data_length )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

    RESET = 0;
    DATA_AQ = 1;
    CRC_COMPARE = 2;
    WAIT = 3;


    persistent state, state = xl_state(RESET,{xlUnsigned, 2, 0});
    persistent crc_enable_internal, crc_enable_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent data_counter, data_counter = xl_state(0,{xlUnsigned, 8, 0});
    persistent crc_reset_internal, crc_reset_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent output_valid_internal, output_valid_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});
    persistent crc_valid_internal, crc_valid_internal = ...
        xl_state(0,{xlUnsigned, 1, 0});

        switch state
            case RESET
                crc_enable_internal = 1;
                crc_reset_internal = 1;
                output_valid_internal = 1;
                crc_valid_internal = 0;


                data_counter = 1;
                if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},data_valid) == 1
                    state = DATA_AQ;
                end


            case DATA_AQ
                if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},data_valid) == 1
                    crc_enable_internal = 1;
                    crc_reset_internal = 0;
                    output_valid_internal = 1;
                    crc_valid_internal = 0;

                    data_counter = xfix({xlUnsigned, 8, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);

                    if xfix({xlUnsigned, 8, 0, xlTruncate, ...
                        xlWrap},data_counter) ≥ xfix({xlUnsigned, 8, 0, ...
                        xlTruncate, xlWrap},data_length-1)
                        state = CRC_COMPARE;
                    end
                else
                    crc_enable_internal = 0;
                    crc_reset_internal = 0;
```

```
47                      output_valid_internal = 0;
48                      crc_valid_internal = 0;
49                  end
50
51
52              case CRC_COMPARE
53                  crc_enable_internal = 1;
54                  crc_reset_internal = 0;
55                  output_valid_internal = 0;
56                  crc_valid_internal = 1;
57
58                  state = RESET;
59
60
61               case WAIT
62                  crc_enable_internal = 0;
63                  crc_reset_internal = 1;
64                  output_valid_internal = 0;
65                  crc_valid_internal = 0;
66
67                  state = RESET;
68
69              otherwise
70                  state = RESET;
71          end
72
73      crc_enable = crc_enable_internal;
74      crc_reset = crc_reset_internal;
75      output_valid = output_valid_internal;
76      crc_valid = crc_valid_internal;
77
78
79
80  end
```

MatLab code for Frame Error Rate calculation state machine

```
function [ N_frame_errors, N_frames] = error_rate_calc( crc_valid, ...
    crc_pass_fail_n, reset )

    IDLE = 0;
    CRC = 1;
    RESET = 2;

    persistent estado, estado = xl_state(RESET,{xlUnsigned, 2, 0});
    persistent n_frame_errors_internal, n_frame_errors_internal = ...
        xl_state(0,{xlUnsigned, 16, 0});
    persistent n_frames_internal, n_frames_internal = xl_state(0,{xlUnsigned, ...
        16, 0});

    if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},reset) == 1
        estado = RESET;
    end

    switch estado

        case IDLE
            disp(5);
            if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},crc_valid) == 1
                estado = CRC;
                n_frames_internal = xfix({xlUnsigned, 16, 0, xlTruncate, ...
                    xlWrap},n_frames_internal + 1);

                if xfix({xlUnsigned, 1, 0, xlTruncate, ...
                    xlWrap},crc_pass_fail_n) == 0
                    n_frame_errors_internal = xfix({xlUnsigned, 16, 0, ...
                        xlTruncate, xlWrap}, n_frame_errors_internal + 1) ;
                end
            end

        case CRC
            if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},crc_valid) == 0
                estado = IDLE;
            end

        case RESET
            if xfix({xlUnsigned, 1, 0, xlTruncate, xlWrap},reset) == 0
                estado = IDLE;
            end

        otherwise
            estado = IDLE;
    end

    N_frame_errors =  n_frame_errors_internal;
    N_frames =  n_frames_internal;
end
```

Appendix B

System Constraints MatLab script file

```
clear all
close all
clc

colorstring = 'rgbcmyk';

%% RS vs Convolutional
SNR = 0:0.1:12;
ber_awgn = berawgn(SNR,'psk',4,'nondiff');
semilogy(SNR,ber_awgn, 'LineWidth', 2.5)
hold on
grid on

SNR = 0:0.1:12;
trellis = poly2trellis(7, [171 133]);
RS = bercoding(SNR, 'RS', 'hard', 255, 213);
Conv = bercoding(SNR, 'conv', 'hard', 0.5, distspec(trellis));

semilogy(SNR,RS,'g', 'LineWidth', 2.5)
semilogy(SNR,Conv,'r', 'LineWidth', 2.5)

legend('QPSK uncoded','RS(255,213)','Convolutional code K=7, R=1/2')
axis([0 12 10^-10 10^-0])
xlabel('SNR (dB)','FontSize',19)
ylabel('Bit Error Rate','FontSize',19)
%title('FEC performance over AWGN channel','FontSize',19)
set(gca,'FontSize',14)

%% QPSK BER over AWGN channel
SNR = 0:0.1:10;
BER_QPSK = berawgn(SNR, 'psk', 4, 'nondiff')
figure()
semilogy(SNR, BER_QPSK, 'LineWidth',2)
hold on
grid on
semilogy(7,BER_QPSK(70+1), '*r', 'LineWidth', 2)
```

```
37
38 strQPSK = strcat('BER = 7.2e-4 @ 7.0dB');
39 text(6.5, BER_QPSK(70+1),strQPSK,'HorizontalAlignment','right', 'FontSize', ...
       12, 'Color', 'k', 'FontWeight', 'bold', 'Margin', 1, 'BackgroundColor', 'w')
40 xlabel('SNR','FontSize', 12)
41 ylabel('BER', 'FontSize', 12)
42 %title('QPSK performance over AWGN channel', 'FontSize', 12)
43 legend('QPSK')
44
45 %% BER_out vs Code efficiency
46 N = 255;
47 K_vector = 1:2:249;
48 R = K_vector./N;
49 figure()
50
51 SNR = 7;
52 for i=1:length(K_vector)
53     K = K_vector(i);
54
55     BER_out(i) = bercoding(SNR, 'RS', 'hard', N, K, 'psk', 4, 'nondiff');
56 end
57
58 semilogx(BER_out, R, 'LineWidth', 2)
59 hold on
60 grid on
61
62 [minimo,indice]=min(BER_out);
63 semilogx(minimo, R(indice), '*r','LineWidth', 2)
64
65 strRScode = strcat('\rightarrow RS(255,',num2str(K_vector(indice)),'); ...
       efficiency=', num2str(R(indice),4));
66 text(minimo*4,R(indice),strRScode,'FontSize', 10, 'Color', 'k', 'FontWeight', ...
       'bold', 'Margin', 3, 'BackgroundColor', 'w');
67
68 xlabel('Bit Error Rate @ output', 'FontSize', 12)
69 ylabel('Code efficiency', 'FontSize', 12)
70 %title('Bit Error Rate vs efficiency', 'FontSize', 12)
71 legend(strcat('SNR = ', num2str(SNR), 'dB'))
72
73
74 %% Efficiency vs FER
75 clear all
76
77 frame_size = [128 256 512 1024 2048 4096];  % frame size in Bytes
78 N = 255;
79 K = 165:2:253;
80
81 k = ceil(log2(N+1));            % Elements dimension
82 M = 2^k;                        % Number of different symbols
83 R = K/N;                        % Code rate
84 t = (N-K)/2;                    % Error correction RS capability
85
86 SNR = 7;
87 EbNo = 10.^((SNR)/10);       % Linear values
88
89 for f=1:length(frame_size)
90     for c=1:length(K)
```

```
91
92          k = ceil(log2(N+1));            % Elements dimension
93          M = 2^k;                        % Number of different symbols
94          R = K(c)/N;                        % Code rate
95          t = (N-K(c))/2;                    % Error correction RS capability
96
97          s = (1-qfunc(sqrt(2*R.*EbNo))).^k;  % Probability of symbol is ...
                correctly received
98          PM = 1-s;                           % Probability of code-word symbol ...
                error
99
100         Pes=0;
101         for i = (t+1):N
102             Pes = Pes + (i*nchoosek(N,i).*(PM.^i).*(1-PM).^(N-i));
103         end
104         BER(f,c) = Pes./N;
105
106         BLER(f,c) = 1-(1-BER(f,c)).^K(c);
107
108         NBLOCKS = ceil(frame_size(f)/K(c));
109
110         FER(f,c) = (1-(1-BLER(f,c)).^NBLOCKS);
111
112     end
113
114 end
115
116 %% Efficiency vs FER plots
117 figure()
118 for i=1:length(K)
119     x(:,i)=(K(i)./N) .* (frame_size./(frame_size+8));
120     semilogy((K(i)./N) .* (frame_size./(frame_size+8)), FER(:,i), '-*')
121     hold on
122     grid on
123
124 end
125 hold on
126
127 plot(get(gca,'xlim'), [ 2.3704e-08  2.3704e-08])
128 %title('Efficiency vs FER Frame ','FontSize', 12)
129 axis([0.65 1 10e-15 2])
130 ylabel('FER','FontSize', 12)
131 xlabel('Total Efficiency','FontSize', 12)
132
133
134 %% Eff vs FER for framesize=256 for all RS codes
135 figure()
136 for i=1:length(K)
137
138     semilogy((K(i)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,i), '-*', ...
            'Linewidth', 1.5)
139
140     hold on
141     grid on
142 end
143
144 plot(get(gca,'xlim'), [ 2.3704e-08  2.3704e-08])
```

```
145 semilogy((K(26)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,26), '-rO', ...
        'Linewidth', 2)
146
147 strRScode = strcat('\leftarrow RS(255,',num2str(K(26)),'); efficiency=', ...
        num2str((K(26)./N) .* (frame_size(2)./(frame_size(2)+8))));
148 text((K(27)./N) .* (frame_size(2)./(frame_size(2)+8)), ...
        FER(2,26),strRScode,'FontSize', 10, 'Color', 'k', 'FontWeight', 'bold', ...
        'Margin', 1, 'BackgroundColor', 'w');
149 hold on
150
151 %title('Efficiency vs FER for Frame Size = 256','FontSize', 12)
152 axis([0.65 1 10e-15 2])
153 ylabel('FER','FontSize', 12)
154 xlabel('Total Efficiency','FontSize', 12)
155
156 %% Efficiency vs BER for one code
157 figure()
158
159 i=25;
160
161     semilogy((K(i)./N) .* (frame_size./(frame_size+8)), FER(:,i), '-*')
162     hold on
163     grid on
164     semilogy((K(i)./N) .* (frame_size(2)./(frame_size(2)+8)), FER(2,i), ...
            '-rO', 'Linewidth', 2)
165
166 axis([0.785 0.85 3e-9 2e-7])
167 ylabel('FER','FontSize', 12)
168 xlabel('Total Efficiency','FontSize', 12)
169
170 %% Auxiliar calculations
171 %% Horas a funcionar sem errros para FER
172 bitrate=24e6; %24Mbits/seg
173 framesize=256;    %em bytes
174 frameporseg=bitrate/(8*framesize);
175 tempo=60*60; %tempo em segundos
176 FER=1/(frameporseg*tempo)
177 %% FER para horas
178 FER=1.95*10^-9;
179 bitrate=24000000; %24Mbits/seg
180 framesize=128;    %em bytes
181 tempo_horas=(1/FER)/(bitrate/(framesize*8))/3600
```

Appendix C

MatLab scripts

Burst errors statistical analysis

```
%%
IDLE = 0;
ERROR = 1;
BURST = 2;
BURST_END = 3;

c=0;
state=IDLE;
burstlengthcount = [];

for i=1:length(simout)
    switch state
        case IDLE
            if simout(i) == 1
                state = ERROR;
            end

        case ERROR
            if simout(i) == 1
                c=c+2;
                state = BURST;
            else
                state = IDLE;
            end

        case BURST
            if simout(i) == 1
                c=c+1;
            else
                state = BURST_END;
            end

        case BURST_END
            burstlengthcount = [burstlengthcount c];
            c=0;
            state = IDLE;
    end
```

```
38 end
39
40 number_of_burst_errors = length(burstlengthcount)
41 average_burst_length=mean(burstlengthcount)
42 max_burst_length=max(burstlengthcount)
43 min_burst_length=min(burstlengthcount)
```

MatLab function for computing BER of Reed-Solomon codes

```
% Reed-Solomon bit error probability vs Eb/No
function ber = rsber_estimation(EbNodb,N,K)
    k = ceil(log2(N+1));            % Elements dimension
    M = 2^k;                        % Number of different symbols
    R = K/N;                        % Code rate
    t = (N-K)/2;                    % Error correction RS capability

    EbNo = 10.^((EbNodb)/10);       % Linear Eb/No values

    s = (1-qfunc(sqrt(2*R.*EbNo))).^k;  % Probability of symbol is correctly ...
        received
    PM = 1-s;                           % Probability of code-word symbol error

    % Symbol error probability when code-word error is made
    Pes = 0;
    for i = (t+1):N
        Pes = Pes + (i*nchoosek(N,i).*(PM.^i).*(1-PM).^(N-i));
    end
    Pes = Pes./N;

    % Conversion between symbol error probability to bit error probability
    %ber = Pes.*((2^(k-1))/(2^k-1));
    ber = Pes./k;   %approximated value for lower BER
end
```

Appendix D

Asynchronous Implementation FSMs

CRC TX FSM

```
function [ data_out_select, crc_enable, crc_reset, output_valid, RE ] = ...
    crc_tx_ctrl( En_data, AE, data_length, reset)

    RESET = 0;
    DATA_READ = 1;
    CRC_RESULT = 2;
    WAIT = 3;

    DATA = 0;
    CRC = 1;

    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET,{xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0,{xlUnsigned, 8, 0});

    switch state
        case RESET
            data_out_select = DATA;
            crc_enable = 0;
            crc_reset = 1;
            output_valid = OFF;
            RE = OFF;

            data_counter = 0;

            if En_data == ON && AE == OFF
                state = DATA_READ;
            end

        case DATA_READ
            data_out_select = DATA;
            crc_enable = 1;
```

```
35              crc_reset = 0;
36              output_valid = ON;
37              RE = ON;
38
39              data_counter = xfix({xlUnsigned, 8, 0, xlTruncate, ...
                    xlWrap},data_counter + 1);
40
41              if xfix({xlUnsigned, 8, 0, xlTruncate, xlWrap},data_counter) ≥ ...
                    xfix({xlUnsigned, 8, 0, xlTruncate, xlWrap},data_length)
42                  state = CRC_RESULT;
43              end
44
45              if En_data == OFF || AE == ON
46                  state = WAIT;
47              end
48
49              if reset == ON
50                  state = RESET;
51              end
52
53
54
55          case CRC_RESULT
56              data_out_select = CRC;
57              crc_enable = 1;
58              crc_reset = 0;
59              output_valid = ON;
60              RE = OFF;
61
62              state = RESET;
63
64          case WAIT
65              data_out_select = DATA;
66              crc_enable = 0;
67              crc_reset = 0;
68              output_valid = OFF;
69              RE = OFF;
70
71              if En_data == ON && AE == OFF
72                  state = DATA_READ;
73              end
74
75          otherwise
76              state = RESET;
77
78              data_out_select= DATA;
79              crc_enable = 1;
80              crc_reset = 1;
81              output_valid = OFF;
82              RE = OFF;
83      end
84
85  end
```

CRC RX FSM

```
function [ crc_enable, crc_reset, output_valid, crc_valid, RE ] = ...
    crc_rx_ctrl( En_data, AF, data_length, reset)

    RESET = 0;
    DATA_READ = 1;
    CRC_COMPARE = 2;
    WAIT = 3;

    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET,{xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0,{xlUnsigned, 8, 0});

    switch state
        case RESET
            crc_valid = OFF;
            crc_enable = 1;
            crc_reset = 1;
            output_valid = OFF;
            RE = OFF;

            data_counter = 0;

            if En_data == ON && AF == OFF
                state = DATA_READ;
            end

        case DATA_READ
            crc_valid = OFF;
            crc_enable = 1;
            crc_reset = 0;
            output_valid = ON;
            RE = ON;

            data_counter = xfix({xlUnsigned, 8, 0, xlTruncate, ...
                xlWrap},data_counter + 1);

            if xfix({xlUnsigned, 8, 0, xlTruncate, xlWrap},data_counter) ≥ ...
                xfix({xlUnsigned, 8, 0, xlTruncate, xlWrap},data_length)
                state = CRC_COMPARE;
            end

            if En_data == OFF || AF == ON
                state = WAIT;
            end

            if reset == ON
                state = RESET;
            end

        case CRC_COMPARE
            crc_enable = 1;
```

```
52              crc_reset = 0;
53              output_valid = OFF;
54              crc_valid = ON;
55              RE = ON;
56
57              state = RESET;
58
59          case WAIT
60              crc_valid = OFF;
61              crc_enable = 0;
62              crc_reset = 0;
63              output_valid = OFF;
64              RE = OFF;
65
66              if En_data == ON && AF == OFF
67                  state = DATA_READ;
68              end
69
70          otherwise
71              state = RESET;
72
73              crc_valid= OFF;
74              crc_enable = 1;
75              crc_reset = 1;
76              output_valid = OFF;
77              RE = OFF;
78      end
79
80  end
```

Reed-Solomon TX FSM

```
function [ RS_data_valid, RS_data_last, RE ] = TX_CH_ENC_RS_CTRL( En_Data, ...
    AE, reset)

    % states declaration
    RESET = 0;
    RS_READ = 1;
    RS_WAIT = 2;
    WAIT = 3;

    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET, {xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0, {xlUnsigned, 11, 0});

    switch state
        case RESET
            RS_data_valid = OFF;
            RS_data_last = OFF;
            RE = OFF;

            data_counter = 0;

            if AE == OFF && En_Data == ON
                state = RS_READ;
            else
                state = WAIT;
            end


        case RS_READ


            if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},212)
                RS_data_last = ON;
                RE = OFF;
            else
                RS_data_last = OFF;
                RE = ON;
            end

            RS_data_valid = ON;

            data_counter = xfix({xlUnsigned, 11, 0, xlTruncate, ...
                xlWrap},data_counter + 1);

            if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},213)
                state = RS_WAIT;
            end

            if En_Data == OFF || AE == ON
```

```
51                  state = WAIT;
52              end
53
54              if reset == ON
55                  state = RESET;
56              end
57
58          case RS_WAIT
59              RS_data_valid = OFF;
60              RS_data_last = OFF;
61              RE = OFF;
62
63              if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                    xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},1074) % 1074
64                  state = RESET;
65              else
66                  data_counter = xfix({xlUnsigned, 11, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);
67              end
68
69              if reset == ON
70                  state = RESET;
71              end
72
73          case WAIT
74              RS_data_valid = OFF;
75              RS_data_last = OFF;
76              RE = OFF;
77
78              if AE == OFF && En_Data == ON
79                  state = RS_READ;
80              end
81
82              if reset == ON
83                  state = RESET;
84              end
85
86          otherwise
87              RS_data_valid = OFF;
88              RS_data_last = OFF;
89              RE = OFF;
90
91              state = RESET;
92
93      end
94
95  end
```

Reed-Solomon RX FSM

```
function [ RS_data_valid, RS_data_last, RE ] = RX_CH_ENC_RS_CTRL( En_Data, ...
    AF, reset)

    % states declaration
    RESET = 0;
    RS_READ = 1;
    RS_WAIT = 2;
    WAIT = 3;

    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET, {xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0, {xlUnsigned, 11, 0});

    switch state
        case RESET
            RS_data_valid = OFF;
            RS_data_last = OFF;
            RE = OFF;

            data_counter = 0;

            if AF == OFF && En_Data == ON
                state = RS_READ;
            else
                state = WAIT;
            end


        case RS_READ


            if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},254)
                RS_data_last = ON;
            else
                RS_data_last = OFF;
            end

            RS_data_valid = ON;
            RE = ON;

            data_counter = xfix({xlUnsigned, 11, 0, xlTruncate, ...
                xlWrap},data_counter + 1);

            if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},255)
                state = RS_WAIT;
            end

            if En_Data == OFF || AF == ON
                state = WAIT;
```

```
51              end
52
53              if reset == ON
54                  state = RESET;
55              end
56
57          case RS_WAIT
58              RS_data_valid = OFF;
59              RS_data_last = OFF;
60              RE = OFF;
61
62              if xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},data_counter) == ...
                    xfix({xlUnsigned, 11, 0, xlTruncate, xlWrap},1074) % 1074
63                  state = RESET;
64              else
65                  data_counter = xfix({xlUnsigned, 11, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);
66              end
67
68              if reset == ON
69                  state = RESET;
70              end
71
72          case WAIT
73              RS_data_valid = OFF;
74              RS_data_last = OFF;
75              RE = OFF;
76
77              if AF == OFF && En_Data == ON
78                  state = RS_READ;
79              end
80
81              if reset == ON
82                  state = RESET;
83              end
84
85          otherwise
86              RS_data_valid = OFF;
87              RS_data_last = OFF;
88              RE = OFF;
89
90              state = RESET;
91
92      end
93
94  end
```

Interleaving TX FSM

```
function [ INT_data_valid, INT_data_last, RE ] = TX_CH_ENC_INT_CTRL( AE, ...
    En_Data, reset )

    % states declaration
    RESET = 0;
    INT_READ = 1;
    INT_WAIT = 2;
    WAIT = 3;


    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET, {xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0, {xlUnsigned, 12, 0});

    switch state
        case RESET
            INT_data_valid = OFF;
            INT_data_last = OFF;
            RE = OFF;

            data_counter = 0;

            if AE == OFF && En_Data == ON
                state = INT_READ;
            else
                state = WAIT;
            end

        case INT_READ
            INT_data_valid = ON;
            INT_data_last = OFF;
            RE = ON;

            if xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) < ...
                xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},2550-1)
                data_counter = xfix({xlUnsigned, 12, 0, xlTruncate, ...
                    xlWrap},data_counter + 1);

            elseif xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) ...
                == xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},2550-1)
                INT_data_last = ON;
                data_counter = 0;
                state = INT_WAIT;
            end

            if En_Data == OFF || AE == ON
                state = WAIT;
            end

            if reset == ON
                state = RESET;
```

```
51              end
52
53          case INT_WAIT;
54              INT_data_valid = OFF;
55              INT_data_last = OFF;
56              RE = OFF;
57
58              % wait for 10 RS codewords (2550 symbols) and ...
                    Interleaving/Deinterleaving latency (5*2=10)
59              if xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) < ...
                    xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},255*10+10-1)
60                  data_counter = xfix({xlUnsigned, 12, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);
61              else
62                  state = RESET;
63              end
64
65          case WAIT
66              INT_data_valid = OFF;
67              INT_data_last = OFF;
68              RE=OFF;
69
70              if En_Data == ON && AE == OFF
71                  state = INT_READ;
72              end
73
74          otherwise
75              INT_data_valid = OFF;
76              INT_data_last = OFF;
77              RE = OFF;
78
79              state = RESET;
80      end
81  end
```

Interleaving RX FSM

```
function [ INT_data_valid, INT_data_last, RE ] = RX_CH_ENC_INT_CTRL( AF, ...
    En_Data, reset )

    % states declaration
    RESET = 0;
    INT_READ = 1;
    INT_WAIT = 2;
    WAIT = 3;


    %Boolean definitions
    ON = xfix({xlBoolean}, 1); % Boolean '1'
    OFF = xfix({xlBoolean}, 0); % Boolean '0'

    persistent state, state = xl_state(RESET, {xlUnsigned, 2, 0});
    persistent data_counter, data_counter = xl_state(0, {xlUnsigned, 12, 0});


    switch state
        case RESET
            INT_data_valid = OFF;
            INT_data_last = OFF;
            RE = OFF;

            data_counter = 0;

            if AF == OFF && En_Data == ON
                state = INT_READ;
            else
                state = WAIT;
            end

        case INT_READ
            INT_data_valid = ON;
            INT_data_last = OFF;
            RE = ON;

            if xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) < ...
                xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},2550-1)
                data_counter = xfix({xlUnsigned, 12, 0, xlTruncate, ...
                    xlWrap},data_counter + 1);

            elseif xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) ...
                == xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},2550-1)
                INT_data_last = ON;
                data_counter = 0;

                state = INT_WAIT;
            end

            if En_Data == OFF || AF == ON
                state = WAIT;
            end

```

```
51              if reset == ON
52                  state = RESET;
53              end
54
55          case INT_WAIT;
56              INT_data_valid = OFF;
57              INT_data_last = OFF;
58              RE = OFF;
59
60              % wait for 10 RS codewords (2550 symbols) and ...
                    Interleaving/Deinterleaving latency (5*2=10)
61              if xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},data_counter) < ...
                    xfix({xlUnsigned, 12, 0, xlTruncate, xlWrap},255*10+10-1)
62                  data_counter = xfix({xlUnsigned, 12, 0, xlTruncate, ...
                        xlWrap},data_counter + 1);
63              else
64                  state = RESET;
65              end
66
67          case WAIT
68              INT_data_valid = OFF;
69              INT_data_last = OFF;
70              RE=OFF;
71
72              if En_Data == ON && AF == OFF
73                  state = INT_READ;
74              end
75
76          otherwise
77              INT_data_valid = OFF;
78              INT_data_last = OFF;
79              RE = OFF;
80
81              state = RESET;
82      end
83  end
```

Data Generator Buffer Control FSM

```
function [En_Data, AE, AF, Cnt_Reset, estado] = ...
    TX_BUFFER_CTRL_Gen_RE(full_percent, RESET)

%FSM state coding
Wait_50 = 0;        %Initial wait for the input buffer to fill to 50%
Enable_Data = 1;    %Allows processing of data
Wait_Empty = 2;     %Wait buffer to empty below threshold
Count_Reset  = 3;      %Counter reseting state

%FSM variable
persistent state,
state = xl_state(0, {xlUnsigned, 2, 0});

%Boolean definitions
ON = xfix({xlBoolean}, 1); % Boolean '1'
OFF = xfix({xlBoolean}, 0); % Boolean '0'

%working variables
%Number of loaded values counter
% persistent n_load_values,
% n_load_values = xl_state(0, {xlUnsigned, 15, 0});

%Generic output definition
if full_percent > 13 % 87.5%
    AF = ON;
else
    AF = OFF;
end

if (state == Enable_Data || state == Wait_Empty) && full_percent > 1 % 12.5%
    AE = OFF;
else
    AE = ON;
end

%FSM state transition
switch state
    case Wait_50
        En_Data = ON;
        Cnt_Reset = OFF;
        estado = 0;
        if RESET == OFF
            if full_percent > 8 % 50%
                state = Enable_Data;
            else
                state = Wait_50;
            end
        else
            state = Count_Reset;
        end

    case Enable_Data
        En_Data = ON;
        Cnt_Reset = OFF;
```

```
54          estado = 1;
55          if RESET == OFF
56              if full_percent > 13 % 87.5%
57                  state = Wait_Empty;
58              else
59                  state = Enable_Data;
60              end
61          else
62              state = Count_Reset;
63          end
64
65      case Wait_Empty
66          En_Data = OFF;
67          Cnt_Reset = OFF;
68          estado = 2;
69          if RESET == OFF
70              if full_percent < 8 % 50%
71                  state = Enable_Data;
72              else
73                  state = Wait_Empty;
74              end
75          else
76              state = Count_Reset;
77          end
78
79      case Count_Reset
80          En_Data = OFF;
81          Cnt_Reset = ON;
82          estado = 3;
83          state = Wait_50;
84
85      otherwise
86          En_Data = OFF;
87          Cnt_Reset = OFF;
88          estado = 3;
89          state = Wait_50;
90  end
91
92  end
```

TX Buffer Control FSM

```
function [En_Data, AE, AF] = TX_BUFFER_CTRL(full_percent,RESET)

%FSM state coding
Wait_90 = 0;         %Initial wait for the input buffer to fill to 50%
Enable_Data = 1;     %Allows processing of data
Wait_Empty = 2;      %Wait buffer to empty below threshold

%FSM variable
persistent state,
state = xl_state(0, {xlUnsigned, 2, 0});

%Boolean definitions
ON = xfix({xlBoolean}, 1); % Boolean '1'
OFF = xfix({xlBoolean}, 0); % Boolean '0'

%working variables
%Number of loaded values counter
% persistent n_load_values,
% n_load_values = xl_state(0, {xlUnsigned, 15, 0});

%Generic output definition
if full_percent > 13 % 87.5%
    AF = ON;
else
    AF = OFF;
end

if (state == Enable_Data || state == Wait_Empty) && full_percent > 1 % 12.5%
    AE = OFF;
else
    AE = ON;
end

%FSM state transition
switch state
    case Wait_90
        En_Data = ON;
        if RESET == OFF
            if full_percent > 13 % 90%
                state = Enable_Data;
            else
                state = Wait_90;
            end
        else
            state = Wait_90;
        end

    case Enable_Data
        En_Data = ON;
        if RESET == OFF
            if full_percent > 13 % 87.5%
                state = Wait_Empty;
            else
                state = Enable_Data;
```

```
55              end
56          else
57              state = Wait_90;
58          end
59
60      case Wait_Empty
61          En_Data = OFF;
62          if RESET == OFF
63              if full_percent < 10 % 67.5%
64                  state = Enable_Data;
65              else
66                  state = Wait_Empty;
67              end
68          else
69              state = Wait_90;
70          end
71
72      otherwise
73          En_Data = OFF;
74          state = Wait_90;
75  end
76
77  end
```

TX Interleaver Buffer Control FSM

```
function [En_Data, AE, AF] = TX_BUFFER_CTRL_INT(full_percent,RESET)

%FSM state coding
Wait_90 = 0;         %Initial wait for the input buffer to fill to 50%
Enable_Data = 1;     %Allows processing of data
Wait_Empty = 2;      %Wait buffer to empty below threshold

%FSM variable
persistent state,
state = xl_state(0, {xlUnsigned, 2, 0});

%Boolean definitions
ON = xfix({xlBoolean}, 1); % Boolean '1'
OFF = xfix({xlBoolean}, 0); % Boolean '0'

%Generic output definition
if full_percent > 10 % 87.5%
    AF = ON;
else
    AF = OFF;
end

if (state == Enable_Data || state == Wait_Empty) && full_percent > 1 % 12.5%
    AE = OFF;
else
    AE = ON;
end

%FSM state transition
switch state
    case Wait_90
        En_Data = ON;
        if RESET == OFF
            if full_percent > 10 % Garante que pelo menos 2730 bytes est o ...
                disponiveis
                state = Enable_Data;
            else
                state = Wait_90;
            end
        else
            state = Wait_90;
        end

    case Enable_Data
        En_Data = ON;
        if RESET == OFF
            if full_percent > 10 % Garante que pelo menos 2730 bytes est o ...
                disponiveis
                state = Wait_Empty;
            else
                state = Enable_Data;
            end
        else
            state = Wait_90;
```

```
53          end
54
55      case Wait_Empty
56          En_Data = OFF;
57          if RESET == OFF
58              if full_percent < 5 % 67.5%
59                  state = Enable_Data;
60              else
61                  state = Wait_Empty;
62              end
63          else
64              state = Wait_90;
65          end
66
67      otherwise
68          En_Data = OFF;
69          state = Wait_90;
70  end
71
72  end
```

RX Buffer Control FSM

```
function [En_Data, AE, AF] = RX_BUFFER_CTRL(full_percent, RESET)

%FSM state coding
Wait_50 = 0;         %Waits for the input buffer to fill to 50%
Enable_Data = 1;     %Allows processing of data

%FSM variable
persistent state,
state = xl_state(0, {xlUnsigned, 1, 0});

%Boolean definitions
ON = xfix({xlBoolean}, 1); % Boolean '1'
OFF = xfix({xlBoolean}, 0); % Boolean '0'

%working variables
%Number of loaded values counter
% persistent n_load_values,
% n_load_values = xl_state(0, {xlUnsigned, 15, 0});

%Generic output definition
if full_percent > 12 %0.75
    AF = ON;
else
    AF = OFF;
end

if full_percent < 4 %0.25
    AE = ON;
else
    AE = OFF;
end

%FSM state transition
switch state
    case Wait_50
        En_Data = OFF;
        if RESET == OFF
            if full_percent > 8 % 50%
                state = Enable_Data;
            else
                state = Wait_50;
            end
        else
            state = Wait_50;
        end

    case Enable_Data
        En_Data = ON;
        if RESET == OFF
            if full_percent < 4 %0.25
                state = Wait_50;
            else
                state = Enable_Data;
            end
```

```
55          else
56              state = Wait_50;
57          end
58
59      otherwise
60          En_Data = OFF;
61          state = Wait_50;
62  end
63
64  end
```

RX Reed-Solomon Buffer Control FSM

```
function [En_Data, AE, AF] = RX_BUFFER_CTRL_RS(full_percent, RESET)

%FSM state coding
Wait_50 = 0;         %Waits for the input buffer to fill to 50%
Enable_Data = 1;     %Allows processing of data

%FSM variable
persistent state,
state = xl_state(0, {xlUnsigned, 1, 0});

%Boolean definitions
ON = xfix({xlBoolean}, 1); % Boolean '1'
OFF = xfix({xlBoolean}, 0); % Boolean '0'

%working variables
%Number of loaded values counter
% persistent n_load_values,
% n_load_values = xl_state(0, {xlUnsigned, 15, 0});

%Generic output definition
if full_percent > 10
    AF = ON;
else
    AF = OFF;
end

if full_percent < 5
    AE = ON;
else
    AE = OFF;
end

%FSM state transition
switch state
    case Wait_50
        En_Data = OFF;
        if RESET == OFF
            if full_percent > 8 % 50%
                state = Enable_Data;
            else
                state = Wait_50;
            end
        else
            state = Wait_50;
        end

    case Enable_Data
        En_Data = ON;
        if RESET == OFF
            if full_percent < 5 %0.25
                state = Wait_50;
            else
                state = Enable_Data;
            end
```

```
55          else
56              state = Wait_50;
57          end
58
59      otherwise
60          En_Data = OFF;
61          state = Wait_50;
62  end
63
64  end
```